IMT School for Advanced Studies, Lucca
Lucca, Italy

Formal Modeling and Liquidity Improvement in DeFi

PhD Program in Systems Science
Track in Computer Science and Systems Engineering
XXXVI Cycle

By

Margherita Renieri

2026

The dissertation of Margherita Renieri is approved.

PhD Program Coordinator: Prof. Alberto Bemporad, IMT School for Advanced Studies Lucca

Advisor: Prof. Letterio Galletta, IMT School for Advanced Studies Lucca

The dissertation of Margherita Renieri has been reviewed by:

Prof. Luca Aceto, University of Reykjavik

Prof. Santiago Escobar, Polytechnic University of Valencia

Prof. Luca Viganò, King's College London

IMT School for Advanced Studies Lucca
2026

Quando hai chiuso la porta un'altra s'apre
Non esistono chiavi o serrature
Né sbarre, catenacci. Basta voltare
Lo sguardo e spingere
Piano con le mani.

Goliarda Sapienza, Ancestrale

Contents

List of Figures

List of Tables

Acknowledgements

I am deeply grateful to all the people who have supported, inspired, and accompanied me throughout this journey. This work is the result not only of several years of research, but also of an intense personal and intellectual path that I would not have been able to complete alone.

First, I would like to express my sincere gratitude to my advisors, Letterio Galletta and Rocco De Nicola, for their invaluable guidance, constant support, and intellectual generosity throughout these years. I am especially grateful to Letterio for his patience in standing by me over these years and for serving as a steady guide through the many rejected submissions and the far fewer acceptances.

I would like to thank the many professors and researchers I encountered along my path. In particular, I am deeply grateful to Massimo Bartoletti, who introduced me to the world of decentralized finance and has been a constant source of inspiration in more than a metaphorical sense. I am also grateful to Alberto Lluch Lafuente, my advisor during my Erasmus experience in Denmark, who welcomed me into a foreign country and played a crucial role in shaping both my personal and professional growth.

My gratitude extends to Gabriele Costa for encouraging my involvement in dissemination activities and for offering me several opportunities to collaborate, which strengthened my sense of teamwork while working on topics that have always fascinated me.

Thanks to Silvia for her constant presence, for standing by me when words were hard to find, and for sharing both panic

and enthusiasm during the discovery of a new continent.

I am also grateful to Vittorio, a luminous thread with whom I have shared the highs and lows of my life since we met, regardless of physical distance.

I am deeply thankful to my friends, near and far, who have stood by me throughout this journey. Thank you for your patience, encouragement, laughter, and for reminding me that there is life beyond deadlines and revisions. Your presence made even the most challenging moments lighter.

Finally, a special thank you goes to my family, my lifelong companions. I am profoundly grateful for your unconditional love, constant support, and unwavering belief in me. You have always been my guiding stars, wherever I am and wherever you are. This achievement would not have been possible without you.

To conclude, I firmly believe that the threads woven throughout this journey were meant to intertwine, and that our paths did not cross by chance. In this spirit, I borrow the words from the Nobel Lecture by Han Kang: "*I would like to express my deepest gratitude to all those who have connected with me through that thread, as well as to all those who may come to do so.* "

Vita

Feb 10, 1996	Born, Macerata, Italy
2015-2018	Bachelor Degree in Computer Science Final mark: 110/110 cum laude University of Camerino, Camerino, Italy
2018-2020	Master Degree in Computer Science Final mark: 110/110 cum laude University of Camerino, Camerino, Italy
Nov 2020- Jan 2025	PhD Student IMT School for Advanced Studies Lucca, Lucca, Italy
Feb 2023 - Jul 2023	Visiting Period Technical University of Denmark, Lyngby, Denmark
Feb 2025 - Present	Research Fellowship in "Modeling and security analysis of protocols for decentralized finance." IMT School for Advanced Studies Lucca, Lucca, Italy

Publications

1. M. Renieri, L. Galletta, A. Lluch Lafuente, A. Junge, and J. Hsin-yu Chiang, "A Netting Protocol for Liquidity-saving Automated Market Makers," *Blockchain: Research and Applications*, In Press, 2024.
2. M. Renieri and L. Galletta, "A Policy Framework for Regulating External Calls in Smart Contracts," in *Software Engineering and Formal Methods*, A. Madeira and A. Knapp, Eds. Cham: Springer Nature Switzerland, 2025, pp. 52–69. ISBN: 978-3-031-77382-2.
3. G. Costa, S. De Francisci, M. Renieri, S. Valiani, and P. Prinetto, "Tackling the Gender Gap in Cybersecurity Education," in *Proceedings of the Technical Symposium on Computer Science Education (SIGCSE TS 2025)*, Pittsburgh, Pennsylvania, USA, Feb. 26–Mar. 1, 2025.
4. G. Costa, S. De Francisci, L. Galletta, C. P. Brogi, M. Petrocchi, F. Pinelli, R. Pizziol, M. Pratelli, M. Renieri, S. Soderi, *et al.*, "Systems Security Modeling and Analysis at IMT Lucca," in *International Symposium on Leveraging Applications of Formal Methods*, Springer, 2024, pp. 13–26.
5. M. Renieri, L. Galletta, A. Lluch Lafuente, and J. Hsin-yu Chiang, "A Netting Protocol for Liquidity-saving Automated Market Makers," in *Proceedings of the 6th Distributed Ledger Technology Workshop (DLT 2024)*, CEUR Workshop Proceedings, vol. 3791, Turin, Italy, May 14–15, 2024. Available: https://ceur-ws.org/Vol-3791/
6. M. Renieri, A. Renieri, and L. Galletta, "WalkthroughCyber: Teaching Cyber-Awareness in Montessori Middle Schools," in *Proceedings of the 5th Workshop on Education, Training and Awareness in Cybersecurity (ETACS 2025)*, Ghent, Belgium, Aug. 11–14, 2025.

Presentations

1. M. Renieri, "WalkthroughCyber: Teaching Cyber-Awareness in Montessori Middle Schools," 5th Workshop on Education, Training and Awareness in Cybersecurity (ETACS 2025), Ghent, Belgium, August 11, 2025.
2. M. Renieri, "Declarative Design in DeFi: A Systematic Model of Intent-Based Protocols," 7th Distributed Ledger Technologies Workshop (DLT 2025), Pizzo (VV), Calabria, Italy, June 12–14, 2025.
3. M. Renieri, "Tackling the Gender Gap in Cybersecurity Education," 56th ACM Technical Symposium on Computer Science Education (SIGCSE 2025), Pittsburgh, Pennsylvania, USA, February 26–March 1, 2025.
4. M. Renieri, "A Policy Framework for Regulating External Calls in Smart Contracts," 22nd International Conference on Software Engineering and Formal Methods (SEFM 2024), Aveiro, Portugal, November 4–8, 2024.
5. M. Renieri, "A Netting Protocol for Liquidity-saving Automated Market Makers," Second Secure Software From First Principles Workshop (SWOPS 2024), SERICS - Spoke 6, Lucca, Italy, June 13–14, 2024.
6. M. Renieri, "A Netting Protocol for Liquidity-saving Automated Market Makers," 6th Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14–15, 2024.
7. M. Renieri, "Detection of De-Fi Profitable Scenarios through History-Based Policies," 6th Distributed Ledger Technology Workshop (DLT 2024), Turin, Italy, May 14–15, 2024.
8. M. Renieri, "Equilibria in DeFi from State Context Inspection," 5th Distributed Ledger Technology Workshop (DLT 2023), Bologna, Italy, May 25–26, 2023.

Awards

- Outstanding Reviewer Award, **SIGCSE Technical Symposium 2025**, 2025.
- Best Paper Award, **6th Distributed Ledger Technology Workshop (DLT 2024)**, for the paper: "A Netting Protocol for Liquidity-saving Automated Market Makers," 2024.
- 2nd Prize, Project Award at **International School on Algorand Smart Contracts**, 2022.

Abstract

Liquidity underpins the efficiency and stability of Decentralised Finance (DeFi). With Decentralized Exchanges (DEXs), new paradigms of provision and trading emerge, making it vital to understand liquidity dynamics, from liquidity-saving mechanisms (LSMs) to aggregators that fuse on- and off-chain depth. This thesis develops three approaches to confronting liquidity challenges. First, we adapt netting LSM to DeFi to minimise the liquidity required by Automated Market Makers execution, introducing a queuing mechanism that selects liquidity feasible sequences and settles them atomically on-chain. Second, we formalize intent-based protocols, where users declare outcomes, solvers compete to realize them, and on-chain settlement verifies correctness in a single workflow. Our model captures the mechanisms related to intent aggregation, solver competition, and trade settlement, addressing liquidity fragmentation, execution inefficiency, and gas costs while enhancing scalability, composability, and user experience. Finally, we present a core calculus for Ethereum smart contracts that adds code-level policy constructs to monitor and enforce desired behaviors during external calls. It detects classic vulnerabilities (e.g., reentrancy) and enforces behavioral properties over execution traces, augmenting static analysis with runtime safeguards. Against the backdrop of DEX price fragmentation, where identical assets trade at different prices across venues, we examine the inefficiencies and vulnerabilities, focusing on liquidity-driven price manipulation attacks (e.g. flash loan arbitrage).

Chapter 1

Introduction

Over the past few decades, finance has transitioned from centralized systems to increasingly decentralized, transparent, and programmable infrastructures. This transformation reflects a shift from traditional *Centralized Finance* toward *Decentralized Finance* (DeFi) [1]. The emergence of DeFi [2] reshapes financial services and the way in which they are structured, delivered, and governed. It redefines how financial services are created and accessed, opening up new opportunities while introducing novel risks. Understanding this transition is essential for designing the next generation of mechanisms that blend the robustness of traditional systems with the flexibility and openness of decentralized networks. Instead of relying on central intermediaries, DeFi deploys protocols as smart contracts on public blockchains that encode financial logic, enabling users to lend, borrow, trade, and earn interest. The core innovation of DeFi lies in its composability and transparency. Protocols can interact through standardized interfaces, and every transaction is publicly verifiable on-chain. The rapid growth of DeFi is witnessed by the total value of crypto assets deposited in DeFi protocols, dApps, or blockchain ecosystems – known as Total Value Locked – user adoption and protocol development, which underscores its potential. At the same time, the open nature of DeFi presents new risks, including smart contract vulnerabilities, governance disputes, and market manipulation. The ab-

sence of centralized structures makes regulatory enforcement and consumer protection more complex. Today, DeFi is increasingly interoperating with existing financial systems, contributing to a more resilient and inclusive global financial architecture. This transition is ongoing as DeFi evolves from imperative, transaction-based interactions toward declarative, intent-driven flows in which users express what they want to achieve rather than how to execute it. At the same time, solver markets are deepening, and execution is increasingly abstracted. Users specify what they want; specialized executors decide how — optimizing across venues, chains, and timing — while cryptographic verification and on-chain settlement preserve trust and minimize risk. Building on precursors like limit orders, auction mechanisms, and batching, the next generation of mechanisms adds privacy-preserving proofs, cross-chain coordination, and an improved user experience, with hybrid models blending compliance and user protections from CeFi with the composability and openness of DeFi.

Liquidity is the ability to trade assets on a decentralized exchange easily and quickly without causing significant price changes. It is a relevant aspect of the DeFi ecosystem because it reflects the market's efficiency and stability. DeFi introduces new paradigms for liquidity provision and trading, using Decentralized Exchanges (DEXs) such as AMM [3] and liquidity pools. It is essential to identify the dynamics that focus on liquidity aspects, such as liquidity-saving mechanisms (LSMs) [4] or aggregators, which combine liquidity from on- and off-chain sources to deliver more favorable execution prices for trades.

In this thesis, we present several contributions that address issues stemming from liquidity. As a first contribution, we adapt the traditional netting LSM to DeFi ecosystem to minimize the liquidity required by AMM services during order execution. We start analyzing the LSMs used in traditional finance and develop an off-chain enqueued mechanism that delays transactions that violate liquidity constraints and, when certain conditions are met, selects a feasible transaction sequence from the queue that fulfills the constraints and executes it atomically on the blockchain. We develop an operational semantics that characterizes the

interactions between users and AMMs and the conditions when the liquidity-saving mechanism is triggered. We explore different application scenarios using a simulator, yielding insights into their practical implications. As a second contribution, we formalize the main intent-based protocol [5], in which users declare outcomes, solvers compete to realize them, and on-chain settlement verifies correctness workflows into a single step. It addresses challenges, such as liquidity fragmentation, inefficient execution, and high gas costs, while enhancing scalability, composability, and user experience. We present an operational model that captures the key processes (such as intent aggregation, solver competition, and trade settlement) using a labeled transition system (LTS), and defines the interactions between users, solvers, and on-chain components. By providing a general and extensible semantic foundation, our work may contribute to the development of a robust, efficient, and user-aligned DeFi infrastructure. As the final contribution, we introduce a formal framework for specifying and enforcing security policies on smart contracts that interact with others on Ethereum via external calls that allow a contract to invoke a function of another contract. We present a core calculus for smart contracts, equipped with constructs for specifying policies at the code level, allowing for the monitoring and enforcement of desired behaviors. We provide the formal semantics of this calculus and describe how our approach detects classical vulnerabilities (such as reentrancy), while also offering a flexible and expressive mechanism for enforcing behavioral properties over execution traces, thereby augmenting static analysis with real-time defense. In DEXs, prices for identical assets can vary across venues, provoking inconsistent responses to market dynamics. This fragmentation introduces inefficiencies and potential attacks, such as price manipulation and oracle exploitation, which can result in either unfair advantages for attackers or significant losses for honest participants. We focus our attention on price manipulation attacks that are strictly related to the liquidity movement among DEXs (such as flash-loan-based arbitrage).

Across these contributions, the choice of transition systems is guided by the specific modeling objectives of each setting. Liquidity-saving mech-

anisms are naturally captured by transition systems that emphasize execution ordering, atomicity, and constraint satisfaction, reflecting how queued transactions evolve toward feasible execution states. In contrast, intent-based protocols are more appropriately modeled using labeled transition systems that explicitly represent interaction, competition, and coordination among heterogeneous agents. Rather than enforcing a single uniform formalism, we deliberately adopt transition systems tailored to the abstraction level and interaction patterns of each mechanism, striking a balance between expressiveness, analytical clarity, and faithfulness to real-world protocol execution.

The thesis is organized as follows. Chapter 2 provides the technical and economic background. Firstly, we introduce DeFi order instructions and DEX architectures, which are formalized in later chapters. We then explain flash-loan mechanics and the main attack vectors built upon them, and finally survey LSMs from traditional finance as tools for handling illiquidity. Chapter 3 presents a netting-based LSM for AMM-driven DEXs, enabling multi-user, multi-step exchanges under liquidity constraints. Chapter 4 surveys intent-based approaches and, starting from a common architectural skeleton, develops a formal model capturing their core components and execution flow. Chapter 5 introduces a policy framework for enforcing security and correctness in smart contracts, detecting issues such as reentrancy and flash-loan-driven arbitrage. Chapter 6 concludes and outlines future directions of the research.

The Origin of the Chapters

Most of the chapters of the thesis are based on already published or submitted papers.

In particular:

- Chapter 3, which presents the netting mechanism for AMMs, is based on:
  - M. Renieri, L. Galletta, A. Lluch Lafuente, and J. Hsin-yu Chiang. *A Netting Protocol for Liquidity-saving Automated Market*

Makers. In: Proceedings of the 6th Distributed Ledger Technology Workshop (DLT 2024), CEUR Workshop Proceedings, Vol. 3791, Turin, Italy.

  - M. Renieri, L. Galletta, A. Lluch Lafuente, A. Junge, and J. Hsin-yu Chiang. *Netting-based Liquidity-saving Automated Market Makers*. Blockchain: Research and Applications, 2025. DOI: 10.1016/j.bcra.2025.100361.

- Chapter 4, which explores the modeling of intent-based protocols for DeFi, is the result of ongoing research currently submitted to blockchain-related venues and workshops.

- Chapter 5, which introduces a policy framework for secure smart contract execution, is based on:

  - M. Renieri and L. Galletta. *A Policy Framework for Regulating External Calls in Smart Contracts*. In: *Software Engineering and Formal Methods*. Eds. A. Madeira and A. Knapp. Springer Nature Switzerland, 2025. DOI: 10.1007/978-3-031-77382-2_4.

These works have been progressively refined, extended, and integrated throughout the development of this thesis to provide a comprehensive and formalized contribution to the analysis and design of secure and efficient DeFi systems.

Chapter 2

Background

This chapter introduces the fundamental concepts for an in-depth understanding of the thesis upon which the subsequent chapters build, providing a self-contained overview of the approaches developed later.

2.1 Orders

In DeFi, an *order* is an instruction to buy or sell a specified quantity of a token under stated price and timing conditions, executed either via a decentralized order book or against an AMM curve. An AMM maintains liquidity in on-chain pools and quotes prices through a predefined mathematical rule, the *curve*, which relates the pool's token balances. Executing an order against an AMM therefore means trading directly with the pool at the price implied by its current reserves, rather than matching with another trader's order. For example, suppose Alice wishes to swap USDC for ETH on an AMM. The interface displays the amount of ETH she would receive based on the pool's pricing rule. Once she confirms the transaction, the smart contract transfers her USDC into the pool, returns the corresponding amount of ETH determined by the curve (e.g., the constant-product rule used by Uniswap-style AMMs), and updates the pool reserves, all within a single atomic on-chain operation. Execution is *non-custodial*: users retain control of their assets, authorize

transactions with their keys, and smart contracts settle trades atomically without a central intermediary. Different order types, in turn, shape how well an order executes in practice. To illustrate this, we introduce the notions of *fill probability* and *slippage profile* at a high level. The *fill probability* of an order refers to the likelihood that it will execute under prevailing market conditions, high for orders that trade directly against an AMM pool, but potentially lower for orders resting on a book awaiting a counterparty. The *slippage profile* describes how much the received execution price may deviate from the expected price due to liquidity conditions or trade size; for AMMs, slippage increases as a trade moves further along the curve and draws down more of the pool's reserves. Order types encode execution behavior (e.g., price limits, time-in-force, triggers) and how liquidity is accessed (book queues vs. pool reserves), thereby shaping both risk management and execution quality. Because objectives differ, such as minimizing costs, locking in profits, or prioritizing speed, traders choose the order type that best aligns with their goals. Different order types yield different fill probabilities, slippage profiles, and fees.

DeFi orders are essential tools for managing execution speed, price precision, and risk exposure, enabling strategies that rival those of centralized exchanges while maintaining transparency and minimizing trust through smart contract enforcement. To provide a clear overview, Table 1[1] summarizes typical benefits, drawbacks, and representative examples for each. Below, we introduce the most relevant order types in DeFi and contrast their behavior. At the highest level, orders are divided into **buy orders**, which instruct the purchase of an asset, or **sell orders**, which instruct its sale.

Market Orders. A *market order* instructs the protocol to execute immediately at the best price available. It is best used when speed and certainty of filling outweigh price precision. On an order-book (see the following section), it consumes quotes from the top of the book and, if needed, walks up/down the ladder until the requested size is filled; on an AMM (see the following section), it trades against the pool along the

[1]The table is inspired by the online article [6].

Table 1: Summary of common DeFi order types, their characteristics, and practical guidance.

Order Type	Characteristics	When to Use	When to Avoid
Market Order	Executes immediately at best available price (slippage)	Critical immediacy	price precision matters or in illiquid pools with high slippage
Limit Order	Executes only at a specified price or better; provides control but no execution guarantee	Important exact entry/exit levels	Needed rapid execution or the market may not reach the set price
IOC (Immediate or Cancel)	Executes as much as possible instantly; cancels unfilled portion	Acceptable partial fills, no leftover exposure is desired	Mandatory full fill/ waiting for completion is acceptable
FOK (Fill or Kill)	Must be fully executed immediately or canceled entirely	Required full execution (large trades, arbitrage)	Acceptable partial execution or thin liquidity
Limit Maker (Post-Only)	Ensures liquidity provision; order cancels if it would match immediately	Earn maker fees or avoid taker fees; providing liquidity at chosen prices	Immediate execution
MIT (Market-if-Touched)	Becomes a market order when a trigger price is reached	Important reacting instantly once a price level is achieved	Highly volatile markets where the triggered market order may execute poorly.
Stop Loss	Sells automatically when price falls below a set threshold	Downside risk and protection portfolios during downturns	Highly volatile markets where temporary dips may trigger unnecessary exits
Take Profit	Closes a position at a specified profit target	To lock in gains without constant monitoring	Continued expecting upward movement beyond the target level
Stop-Limit / Take-Profit Limit	Triggered at a price, but executes only as a limit order	Required trigger level and precise execution price	Fast markets with the prevention of the entirely limit condition execution
OCO (One Cancels the Other)	Pairs two linked orders; Execution of one cancels the other.	Managing risk and reward without active monitoring	Low liquidity markets where both orders might risk poor execution
OTO (One Triggers the Other)	Secondary order activates only after the primary order executes	Chained strategies	Secondary trade is important regardless of the first order's outcome
OTOCO	Combines OTO and OCO: primary order triggers a conditional pair	Complex strategies requiring both profit-taking and risk control after entry	Simple execution suffices, avoiding unnecessary complexity
Bracket Order	Places stop-loss and take-profit orders once a main order is executed	Hedging a position with both a profit target and risk cap	Volatile market conditions and both triggers may fail
Iceberg Order	Only part of a large order is visible in the book; hides full size	Large trades to minimize market impact	In AMMs where liquidity visibility is key and iceberg behavior is less relevant

pricing curve, moving the marginal price. Market orders typically guarantee prompt execution given sufficient liquidity, but expose the trader to slippage, the gap between the expected price of a trade and the actual execution price, which increases with order size, market volatility, and limited depth. In practice, the *last traded* price can be stale or uninfor-

mative during fast markets or in thin trading pairs[2]; what matters for a market order is the current best bid/ask (order book) or the pool's spot price (AMM) at settlement, which may differ from the quote seen at submission time due to block latency. Many interfaces include a maximum slippage safeguard to prevent fills that are far from expectations.

Limit Orders. A *limit order* specifies a price constraint on execution: a *buy limit* sets the maximum price the user is willing to pay, and a *sell limit* sets the minimum price the user is willing to accept. If the order fills, it will do so only at the limit price or better; otherwise, it rests until the market reaches that level (or the order expires or it is canceled). Limit orders provide tighter control over execution price and can reduce slippage or fees, but they do not guarantee execution and may fill only partially. They are used when the user believes the price will improve or when she prefers price certainty over immediacy. Time-in-force settings and partial-fill policies further tailor their behavior.

Figure 1: Example of standard order types using a thinkorswim trading chart. The annotated sell limit, buy limit, and market levels illustrate the differing execution conditions associated with each order type, contrasting price-bounded limit orders with the immediacy of market orders.

[2]A thin trading pair is a market with limited liquidity and participation, where trading activity is sparse and available volume at quoted prices is small. As a result, bid–ask spreads are typically wide, price movements can be abrupt even for modest trades, and transaction prices may lag underlying supply and demand. These conditions are common in niche or lightly traded assets and lead to higher trading costs, greater execution risk, and weaker price discovery.

To make the distinction concrete, consider Figure 1. A trader who wants to purchase (or sell) the stock as quickly as possible can place a market order, which will be executed immediately at or near the stock's current price (represented by the white line). The market was open when the order was placed and barring unusual market conditions. In this example, the last trade price was $\$149.50$. We now examine the scenarios for each order type. In a *market order*, a market buy (or sell) instructs the venue to execute now at the best available quote. A market buy will lift the best ask and, if insufficient size is available, continue the ask until the requested quantity is filled; the realized execution price is the volume-weighted average of the consumed quotes and may deviate from the last trade due to slippage. In fast or thin markets, the *last* can be stale; what matters for a market order is the *current* best bid/ask and the depth available at the time of execution. By contrast, a *buy limit order* at $\$144$ will fill only at $\$144$ or better (green line); if the market never trades down to that level, the order stays and expires per its time-in-force. If liquidity is available when $\$144$ is touched, the order may execute at $\$144$ or even slightly below (price improvement). Conversely, a *sell limit order* at $\$164$ fills only at $\$164$ or better (red line); absent a rally to that level, it remains unfilled. If sufficient demand appears at or above $\$164$, the order can execute at $\$164$ or higher. However, even if the stock reached the specified limit price, the order may not fill because there may be orders ahead of it. In that scenario, there may not be enough sellers or buyers willing to sell or buy at that limit price, so the order wouldn't fill. Generally, limit orders are executed on a first-come, first-served basis. For this reason, an order could fill at an even better price. For instance, a buy order could execute below a limit price, and a sell order could execute for more than a limit price.

Stop Orders. A *stop order* (or *stop market order*) is an instruction to buy or sell that activates only when a specified *stop price* is achieved. A buy stop triggers above the current price, while a sell stop triggers below the current price. Once these conditions are triggered, the order becomes a live market order and typically fills at the next available price; if the stop

price is never reached, no execution occurs. It is essential to note that fills are not guaranteed near the stop because gaps, volatility, or thin liquidity can result in slippage. A common variant is the *stop limit order*, which, once triggered, submits a limit order to control execution more precisely, at the cost of possible non-execution if the market skips the limit.

It is convenient to use stop orders in different scenarios. For instance, to protect gains on an existing long position, a trader can place a sell stop below the current price, so if a pullback becomes a reversal, the position exits and locks in part of the unrealized profit. To cap downside after entry, a protective sell stop can be set at a predefined risk threshold so the position is closed automatically if the market moves against the trader. And to enter on momentum, a buy stop placed above a resistance or trigger level ensures participation only if the price breaks out, avoiding premature entries while capturing potential continuation. To make these notions more concrete, consider the following example that illustrates how stop orders can be used simultaneously for risk control, profit protection, and momentum-based entry. Suppose a trader buys at $150. To limit potential losses, she places a protective sell stop at $145. If the price rises, she can move this stop upward to secure part of the gains. Meanwhile, if she is watching a resistance level near $160, she may place a buy stop just above it to enter an additional position only if the price breaks out.

2.2 Decentralized Exchanges

Exchange models define how orders are submitted, matched, and settled on-chain, enabling trades executed directly by smart contracts. Decentralized exchanges (DEXs) implement these models to provide non-custodial asset swaps with on-chain settlement and, by Total Value Locked, represent the largest segment of DeFi. They enable users to trade cryptocurrencies, i.e., digital assets with account balances stored on a blockchain, without relying on a third trusted party to facilitate the exchange. In general, DEXs employ two mechanisms that embody different approaches of price discovery, risk sharing, and market access: *automated market makers*

(AMMs) and *order books*. AMMs are reserves of two or more tokens that act as the counterparty to every trade. Prices are set algorithmically by the AMM's pricing rule, so swaps execute against pooled liquidity, enabling permissionless market making by anyone who supplies assets. In contrast, order books are a traditional exchange approach adopted by centralized crypto venues, which match discrete bids and asks submitted by traders.

2.2.1 Automated Market Makers

Automated Market Makers (AMMs) are on-chain mechanisms that provide continuous liquidity for tradable assets at algorithmically determined prices. Their objective is to facilitate trading and, in many designs, to extract profit from the difference between the highest buy offer and the lowest sell offer. Unlike traditional exchanges that rely on order books of bids and asks, AMMs implement DEXs logic directly in smart contracts to facilitate trading, providing liquidity and price discovery in a decentralized manner. They enable users to trade crypto assets at any time by interacting with liquidity pools, rather than matching with a specific counterparty, allowing individual traders to buy and sell without the need for centralized intermediaries. Through liquidity pools, they enable traders to swap tokens at prices determined algorithmically by the relative balances of the tokens in the pool. Liquidity is supplied to the DEX pools by liquidity providers, and liquidity events relate to the addition or removal of tokens to/from a liquidity pool.

In 2018, with the Uniswap protocol [7] introduced the first AMM model, which enabled users to trade ERC-20[3] tokens directly from their wallets and demonstrated the potential for using algorithms for trading. Since then, AMMs have gained significant traction. ERC-20 defines how tokens are created, transferred, and approved for spending, and encodes governance/voting rights and supports fast, programmatic transactions within Ethereum-based systems. In 2020, several platforms, such as SushiSwap [8]

[3]ERC stands for *Ethereum Request for Comments* and represents the best practices for smart contracts on Ethereum.

and PancakeSwap [9], emerged that provide this service, offering additional features and incentives for liquidity providers. Over time, AMMs have evolved to include features like dynamic pricing, multi-asset pools (Balancer [10]), concentrated liquidity (Uniswap V3 [11]), and improved user interfaces. New models, such as Curve [12], focus on stablecoin trading, optimizing for low slippage and high efficiency. The evolution of AMMs has impacted the cryptocurrency landscape, providing users with more accessible and efficient trading options.

To provide liquidity to markets, AMMs require LPs to deposit tokens in liquidity pools. In return, LPs receive rewards in the form of tokens, which correspond to a share of the trading fees charged by an AMM for market making. The amount of rewards results from the trading fees collected (typically 0.3%) and the proportion of an LP's liquidity relative to the total liquidity used to settle trades. In most AMMs, the token price for a *swap* is set internally by the pool's pricing invariant (e.g., the constant-product formula). However, some designs also consult external price oracles to support additional logic such as collateral valuation, liquidations, and safety checks or to incorporate oracle-aware pricing directly. Oracles act as intermediaries that securely relay information from the external world to on-chain smart contracts, providing the prices, and ensuring that blockchain-based applications can interact meaningfully with the real world and other blockchain networks such as market prices, weather conditions, or on-chain information from other networks, enabling smart contracts to execute based on off-chain data. They transform external data into a format that smart contracts can process. Whether it's price feeds, sports scores, or cross-chain metrics, oracles retrieve, validate, and deliver this data in a way that allows smart contracts to execute transactions and complex operations seamlessly. In doing so, oracles empower DeFi protocols by expanding the range of data accessible to smart contracts.

An AMM specifies an algorithmic relationship between pool reserves and swap prices. The most common class is the *constant-function market maker* (CFMM). A CFMM holds reserves supplied by LPs. Traders submit an amount of tokens to receive another in return; if the trade is

accepted, the input amount is added to the reserves, the output basket is removed, and a small fee accrues to the LPs. Acceptance is defined by the *invariant*: after accounting for fees, the constant function evaluated at post-trade reserves must equal its pre-trade value (i.e., the value of the trading function is held constant).

With two assets, CFMMs reduce to a scalar tradeoff curve, which directly maps the input amount to the output amount. This kind of AMM can manage more than two assets (e.g., Balancer), in that case the choice set becomes multi-dimensional: many input amounts may correspond to a given output basket (and vice versa), so traders optimize over amounts that best match their preferences or utility.

Design choices materially affect capital efficiency. *Uniswap v2* spreads liquidity uniformly over the full price line from zero to infinite value, leaving most liquidity idle for correlated pairs. *Uniswap v3* introduces *concentrated liquidity*, allowing LPs to allocate capital to chosen price ranges, delivering the same depth with less capital (and potentially higher fee income) while exposing LPs to range-specific risks.

2.2.2 Order Books

Order books are a traditional method of trading assets and are widely used by centralized exchanges in both the cryptocurrency and traditional financial markets. An order book shows, in real time, every price level's live buy (bid) and sell (ask) orders for a given asset as submitted by traders. Its transparency provides insights into market depth, price trends, and potential resistance or support levels. It reflects the actual levels of supply and demand, enables price discovery, and facilitates informed trading decisions. Moreover, it reveals the market depth: the deeper the book, the more likely trades will execute at or near the quoted price with minimal slippage. Maintaining robust liquidity helps keep prices fair, reduces disruption from large orders, and improves overall market stability – delivering a smoother, more reliable trading experience for users. As shown in Figure 2, order books are typically rendered vertically with *buy* orders (bids) in green at the bottom and *sell* orders (asks) in red at the

Figure 2: Coinbase advanced trading view: a vertically split order book with red asks at the top and green bids at the bottom, both streams scrolling toward the center price.

top, both stacks converging toward the current top of the book. On large venues, the display updates continuously as orders arrive, cancel, or fill. Entries are sorted by price (and then by time), so a quick vertical scan reveals the best bid and best ask, the bid–ask spread (best ask minus best bid), and the mid-price $m = \frac{1}{2}(\text{best bid} + \text{best ask})$, a common reference for quotes and analytics. The cumulative size shows how much a trader can buy or sell at the displayed prices before the available orders are used up, and the execution must continue at worse prices, causing the market price to shift. This is why it serves as an important measure of potential slippage for large trades. The manual trading model relies on third parties known as market makers for trade completion. These traders or institutions supply liquidity by continuously posting bid and ask quotes, replenishing depth, and tightening spreads. By standing ready to trade,

they improve fill probability, reduce slippage, and maintain market liquidity even during periods of volatility. In return for their services, they often benefit from lower trading fees or other incentives offered by exchanges.

An *order-book* operates through a specific sequence: traders submit buy/sell orders with price and size, which are recorded on-chain (or via a hybrid relayer); a matching engine ranks orders by price and time priority and pairs compatible bids and asks. In this model, a trade executes exactly when a buy order matches a sell order, and the match price becomes the current market price. Because traders can specify the exact price at which they wish to buy or sell, the system offers high-precision execution. The visible book drives price discovery: its depth and imbalances move quotes, while its transparency reveals market depth, emerging price trends, and potential support/resistance levels. Once matched, a smart contract atomically settles the trade: transfers assets, updates balances, and changes the book. With sufficient depth across price levels, trades clear quickly with minimal slippage; throughout, users retain self-custody, and smart contracts enforce transparent, tamper-resistant settlement on the blockchain. When a trader, known as *maker*, places a *limit order*, it is added to the book and *provides* liquidity. By contrast, submitting a *market order* to transact immediately at the best available price *consumes* liquidity – the trader is a *taker*.

For instance, a market buy for 1 ETH[4] will consume liquidity on the ask side starting at the *best ask* and continue up through higher price levels until the full 1 ETH is filled; the execution price is then the volume-weighted average of the consumed quotes (and may deviate from the mid due to *slippage* if depth is thin). Limit orders mitigate slippage by letting traders set a maximum buy price (or minimum sell price), but they trade off immediacy – they may fill only partially or not at all until the market reaches the specified level. A fully visible order book enhances decision quality and facilitates price discovery: when quotes and depth are public, competitive quoting and arbitrage quickly drive prices toward fair value. Fine-grained order types (limits, stops, condition-

[4] ETH is the native cryptocurrency of the Ethereum platform.

als) provide precise execution control, supporting risk management and strategy design. When books are deep, competition among LPs tightens spreads and reduces slippage; large trades can be absorbed across multiple levels with minimal price impact. These strengths, however, stem from mechanics that also impose costs. Execution is liquidity-dependent: without opposing orders at target prices, trades must wait or incur slippage. Liquidity often fragments across venues, so the best price or depth may reside elsewhere, raising search and routing complexity. Public quotes and finite latency introduce information leakage and opportunities for Maximal Extractable Value (MEV) [13], defined as the maximum profit that a transaction sequencer or other fast actor can extract by strategically reordering, inserting, or censoring transactions within a block, and for *front-running*, a specific form of MEV in which an actor observes a pending transaction and submits its own transaction ahead of it to profit from the anticipated price or state change, as faster actors can reorder or insert transactions around visible intent. Microstructure complexity (spreads, priority, queue dynamics) adds a learning curve and penalizes naive order placement. Operationally, centralized books deliver throughput but entail custody and regulatory exposure, whereas on-chain books preserve self-custody and auditability at the expense of gas costs and limited constraints that can widen spreads and amplify slippage during congestion.

2.2.3 Comparing AMMs and Order Books

Order books and AMMs implement two distinct execution models. AMMs prioritize simplicity and accessibility: traders swap directly against pooled liquidity at algorithmically determined prices, so trades clear continuously without matching counterparties. This is attractive for long-tail assets and thin markets. On the other hand, order book models provide a higher level of control and precision for traders. They are suited for high-volume trading and offer transparency in terms of market depth and price formation. Experienced traders often prefer order book models due to the strategic depth they offer, including the ability to place

complex order types like stop-loss or limit orders. AMMs can exchange multi-asset pools in a single atomic transaction, it avoids partial-fill risk; it is also storage- and compute-efficient, which can translate into lower on-chain costs. The trade-offs are *price slippage* when pools are shallow and *LP risk* (e.g., divergent/impermanent loss), and pricing that may lag external markets unless anchored by robust oracles. Order books, by contrast, offer fine-grained control and transparency. They expose depth, follow price/time priority, and support limit or stop order types, making them well-suited to high-volume venues and experienced traders. However, thin books can match slowly, incur higher adverse selection, and remain vulnerable to tactics such as spoofing. Liquidity is typically supplied by dedicated market makers with significant capital and strategy, whereas AMM liquidity is crowdsourced from LPs who earn fees pro rata.

Despite these differences, both AMM and order book models are integral to the cryptocurrency trading ecosystem. It is possible to have hybrid solutions: aggregators route across AMMs and order books; AMMs may consult oracles to align with broader markets; and batch/auction layers can further improve price discovery and reduce MEV and execution risk.

2.3 Flash loans

In traditional finance, a *loan* enables a borrower to obtain funds today and repay them over time, typically with interest. In DeFi, lending is implemented through smart contracts. Borrowers usually lock crypto collateral exceeding the over-collateralized loan value; if its value drops below set thresholds, the position is liquidated to repay the debt. Interest rates are algorithmically set by supply–demand dynamics. Under-collateralized lending is a central objective in DeFi, as it allows users to borrow without locking collateral of greater value than the loan itself. Achieving this in a permissionless environment is difficult because market participants are pseudonymous, asset prices are highly volatile, and there is no legal recourse in the event of default. As a result, most lend-

ing protocols rely on over-collateralization, in which the borrower's collateral is escrowed on-chain and individual creditworthiness is largely irrelevant. To fill this gap while maintaining decentralization, current research and practice are exploring on-chain credit primitives, decentralized identity and attestation systems, and lending structures that reference verifiable off-chain collateral.

Flash loans are often mentioned in this context and are financial instruments that allow users to borrow assets without requiring upfront collateral, leveraging the atomicity of blockchain transactions. This atomicity ensures that the entire transaction either completes successfully or is aborted. Intuitively, they work as follows. At the beginning of a transaction, a user requests a loan of some assets to a DeFi service offering flash loans. The user performs various financial operations with the borrowed assets, including arbitrage and collateral swaps, which can be completed within the same atomic transaction. At the end of the transaction, the user pays back the loan plus a small fee. If the user cannot repay the loan, the transaction fails and reverts: all actions taken during the transaction are undone as if they never happened. This mechanism is risk-free for the lending contract and it is possible due to the Ethereum Virtual Machine's (EVM) ability to revert state changes. Qin *et al.* [14] provide a detailed introduction to flash loans, their common uses, and potential attacks facilitated by them.

Listing 2.1 shows a prototypical implementation of a `flashLoan` function, similar to the one of Aave protocol [15].

Listing 2.1: Pseudocode for a `flashLoan` function illustrating liquidity provision and repayment logic.

```
function flashLoan(receiverAddress, asset, amount, params) {
    // Calculate fees for the asset
    // Check if there is enough liquidity
    if (liquidity[asset] < amount) {
        revert("Insufficient liquidity");
    }

    // Update the liquidity to reflect the loan
    liquidity[asset] -= amount;

    // Transfer the assets to the receiver
    transfer(asset, receiverAddress, amount);

```

```
14      // Execute the operation in the receiver contract
15      bool success = receiverAddress.executeOperation(asset, amount, fee,
            msg.sender, params);
16      if (!success) {
17          revert("FlashLoan execution failed");
18      }
19
20      // Collect the borrowed amount plus fee
21      uint256 totalDebt = amount + fee;
22
23      // Check if receiver has enough tokens to repay the loan
24      uint256 receiverBalance = getBalance(receiverAddress, asset);
25      if (receiverBalance < totalDebt) {
26          revert("Insufficient token balance to repay the loan");
27      }
28
29      transferFrom(receiverAddress, self, totalDebt, asset);
30      liquidity[asset] += totalDebt;
31
32      // Emit the flash loan event
33      emit FlashLoan(receiverAddress, asset, amount, fee);
34  }
```

The function above allows a calling smart contract to borrow an asset, execute arbitrary code using that asset via the external call mechanism, and then return the borrowed amount with a small fee. The parameter `receiverAddress` specifies the contract that receives the borrowed assets and executes a custom code via the function `executeOperation` called within the `flashLoan` function. The parameter `asset` contains the address of the asset to be borrowed, while `amount` specifies its quantity. The parameter `params` provides additional data required by the receiver contract for its operations. Once the fees for the borrowed asset are calculated and some sanity checks verify that the pool has enough liquidity, the requested assets are transferred to the receiver contract. Then, the function `flashLoan` invokes the external function `executeOperation`, passing as parameters the borrowed asset, the amount, the requested fees, the caller's address (`msg.sender`), and additional parameters. The receiver contract returns true to signal successful execution; otherwise, the transaction raises an error. After the receiver contract executes its code, it must repay the borrowed amount plus the fees. When the debt is paid back to the lending contract, the liquidity pool is updated to reflect the repayment, restoring the previous amount of assets.

Table 2: Key DeFi attack vectors enabled by flash loans.

Attack vector	Description	Flash loan role	Mitigations
Price oracle manipulation	Temporarily move a DEX price to mislead protocols that read it as an oracle; profit from distorted valuation	Borrow large, short-lived liquidity to push AMM price; unwind after extracting value	Robust oracles, circuit breakers, bounds
Pump & arbitrage (PA&A)	Creation of a market price gap, then arbitrage across venues in the same tx	Fund the pump leg and the cross-venue arb atomically	Depth-aware routing limits, slippage caps, oracle checks pre-settlement
AMM **reserve manipulation**	Cut pool reserves to affect downstream protocols (collateral valuation, liquidation thresholds, mint/burn rates)	Temporary reserve shift using large loans; revert post-settlement	Avoid spot AMM state as oracle; add buffers, TWAP, sanity checks
Wash trading (volume inflation)	Inflate on-chain volume to game rankings/rewards or trigger thresholds	Rent capital to self-trade at scale within one tx	Anti-sybil/anti-wash rules, fee design, detection
Liquidation gaming	Push prices to force liquidations and capture liquidator rewards	Short-term capital to push past thresholds, then liquidate	Oracle hardening, grace periods, auction safeguards

Flash loan attacks uses temporary liquidity to distort on-chain prices or reserves, trigger faulty logic, extract value before the block settles. In general, in this kind of attack vector, an attacker firstly borrows at scale, then manipulates state by twisting price oracles, stressing AMM curves, or abusing brittle liquidation logic, and finally repays the loan in the same transaction while stealing the surplus. Table 2 summarizes the most common flash loan attack patterns, consistent with the taxonomy presented by Qin *et al.* [16], including the attack vectors, real-world examples, and corresponding mitigation strategies. The consequences range from protocol reserve drains and user losses to wider market dislocations and erosion of trust. As Table 2 shows, effective defenses combine robust data and cautious execution by using multi-source, manipulation-resistant oracles; applying conservative slippage and sanity checks; adding circuit breakers and rate limits; and protecting liquidations with auctions.

2.4 Liquidity-Saving Mechanisms

Payment and settlement systems are classified by the transactions they process: payment systems handle *customer transfers*, settlement systems process *inter-bank transfers*, or a combination of both. Inter-bank settlement transfers funds between different bank accounts, typically are routed through a central clearing house or payment system, and underpin most cashless activity, representing settlement for batches of individual payments.

Payment instructions follow two models: *discrete clearing*, in which instructions are accumulated and settled at scheduled intervals, and *real-time systems*, in which instructions are executed continuously. The main purpose is to ensure an efficient turnover of settlements to balance payments and minimize funds movement.

Nowadays, non-cash transaction volumes have increased, and the demand for real-time processing has intensified, introducing the adoption of advanced information technologies, electronic inter-bank platforms, and continuous operating mechanisms.

Real-time systems can be grouped into two broad categories: *continuous net settlement* (CNS), typically privately operated, where payments are posted intraday but final settlement in central-bank money occurs at end-of-day; and *real-time gross settlement* (RTGS), run by central banks, where each payment is settled individually with immediate finality on central-bank accounts. RTGS is the core infrastructure for inter-bank payments. As long as the sending bank has sufficient liquidity, the payment instruction is posted instantly to the receiver's central-bank account; otherwise, the instruction is queued and released automatically once funds become available. The advantage of RTGS systems is that they eliminate the credit risks and the potential risk associated with other types of payment systems. A drawback of RTGS systems is that the liquidity requirements increase when the payments are settled individually in real time. Mobilising the required liquidity imposes costs on the banks, and all other things being equal, the banks have an incentive to economise on liquidity and may prefer to wait for incoming payments before sending

their own payments. This can lead to settlement delays and gridlocks, situations where queued payments mutually block one another because no participant can settle unilaterally. In true real-time processing, liquidity demand is defined by the incoming payments, but it can be smoothed by deferring some instructions and netting queued payments against opposing flows. This insight led to hybrid designs that blend RTGS with deferred-net features to preserve safety while economising on liquidity. Most large-value systems are RTGS at their core, yet increasingly incorporate liquidity-saving features to optimise usage and resolve gridlocks.

This insight led to the development of hybrid architectures that layer deferred-net features onto RTGS, thereby preserving safety and economizing on liquidity. Most systems integrate RTGS with LSMs [17] to optimize usage and resolve stunting scenarios such as gridlock. An LSM performs partial offsetting of queued payments, directly reducing the liquidity required for those offsets; however, by altering banks' timing incentives, it can also encourage strategic withholding, lengthen queues, and increase aggregate liquidity needs despite local savings. Figure 3

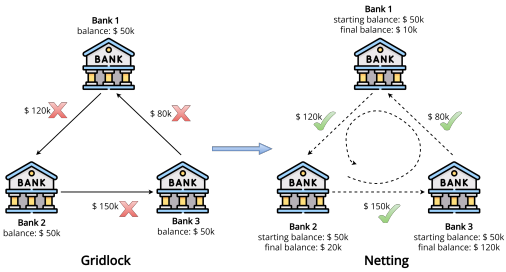

Figure 3: Representation of a *gridlock* scenario and *netting* mechanism.

illustrates a typical gridlock: on the left, Banks 1–3 cannot settle their outgoing payments individually due to insufficient funds, so instructions queue and block one another. On the right, we introduce the *netting* LSM [17], which enables the queue to set the global liquidity level, searching for bilateral or multilateral offsetting cycles and releasing the largest feasible subset that settles without overdrafts. Release can be triggered by incoming funds crossing a threshold, the appearance of an

offsetting payment elsewhere in the queue, or pre-set bilateral/multilateral limits. By partially netting queued payments while respecting bank-defined priorities, LSMs reduce intraday liquidity needs, shorten delays, and resolve gridlock with potential behavioural side effects if participants strategically withhold payments to await offset. Finally, the rise of blockchain ecosystems has spurred efforts to build decentralized inter-bank payment networks (such as the Jasper-Ubin Project [18]) that remove single points of failure and deliver immediate final settlement. These projects adapt RTGS-style to distributed settings without integrating LSMs. Absent a trust-minimized analogue of decentralized multilateral netting, such systems struggle to match the liquidity performance of central–bank–operated hybrids. The adaptation of this mechanism remains a significant open research and engineering challenge, encompassing privacy-preserving queue visibility, consensus on offsetting sets, governance of release rules, and cross-border compliance and interoperability.

2.5 Transaction Evolution Concept

Throughout this thesis, the concept of a transaction evolves significantly in line with the broader transition from traditional blockchain mechanisms to intent-centric architectures. It is essential to clarify how this progression is structured across the various chapters of the work.

In Chapter 5 we adopt transaction as used in conventional blockchain protocols. Here, it is understood as a low-level, imperative instruction explicitly describing an action (e.g., transfer, swap, or deposit) that a user wishes to execute on-chain. This reflects the traditional model in most existing DeFi and blockchain infrastructures.

Chapter 3 introduces a critical intermediate step in re-conceptualizing this model. While still grounded in the transactional format, the netting mechanism begins to shift attention away from single, isolated user instructions and toward a more aggregated, system-level perspective. In this framework, single transactions may be temporarily infeasible due to liquidity constraints, but can later be fulfilled as part of an enqueued

execution. This step begins to weaken the imperative paradigm, emphasizing execution conditions and outcomes over strict, immediate user-specified actions.

The transition is fully realized in Chapter 4. This marks a complete transition away from the imperative paradigm. Instead of requiring users to specify how an operation must be performed, the intent-based paradigm allows them to declare *what* they want to achieve without needing to detail the exact steps involved. The responsibility for determining the optimal execution strategy is delegated to a third party who constructs the most efficient set of transactions to satisfy the user's declared goal.

This progression, from explicit, user-defined transactions to high-level, goal-oriented intents, represents a fundamental change in how financial interactions are modeled. Chapter 3 functions as a transitional stage between these paradigms, whereas the intent-based framework constitutes a complete conceptual break from traditional transaction models, advancing a declarative and user-driven architecture inspired by recent developments.

Chapter 3

Netting

In DeFi systems, AMMs are DEXs that facilitate the trading of crypto-assets through the use of token reserves. Users engaging with these exchanges are subject to fixed trading fees as well as gas fees imposed by the underlying blockchain network. As in traditional financial systems, users are generally required to provide an upfront amount of crypto-assets as collateral in order to access decentralized financial services.

In protocols such as Uniswap, interactions with an AMM are generally restricted to individual, atomic operations. As these systems depend on liquidity providers who deposit tokens into smart contracts, maintaining sufficient liquidity is critical. Without it, the economic performance of the protocol may deteriorate, reducing its efficiency and discouraging user participation.

In this chapter, we introduce a novel LSM for AMMs that allows users to operate without upfront token balances. Inspired by *netting* [19, 20, 21], a technique from traditional finance used to offset payment obligations, our LSM maintains a transaction queue, identifies feasible subsets of pending operations, and atomically settles them once liquidity constraints are satisfied. This is possible by permitting temporary overdrafts, which are later resolved through a netting process that aggregates transactions over a defined time window. By removing the strict requirement for immediate liquidity, our mechanism addresses a key limitation

of current DeFi protocols and enables complex, multi-party, and multi-AMM arbitrage strategies that would otherwise be infeasible.

Here, we formalize and illustrate the design of our LSM using a two-level operational semantics as a labeled transition system (LTS). The first level captures standard interactions between users and AMMs, while the second models the behavior introduced by our mechanism, including the queuing of transactions that violate liquidity constraints and the conditions under which the netting process is triggered. This layered approach allows us to express complex dynamics and support more realistic scenarios that extend beyond the limitations of current AMM models.

Also, we demonstrate that in specific settings, users can successfully increase their token reserves or complete arbitrage sequences even when they lack initial liquidity, highlighting the expressive power and flexibility of our model compared to traditional AMMs.

Additionally, we present a polynomial-time netting algorithm capable of selecting executable subsequences of actions while maintaining liquidity constraints. The algorithm handles interdependencies among transactions and includes a variant that prioritizes actions based on their effects on wallet balances. We also examine an off-line version of the netting mechanism, which raises important questions regarding the design of incentive mechanisms to cover gas costs and ensure participation.

To validate the feasibility of our approach, we implement an open-source Haskell simulator [22] which supports experimentation with a range of trading scenarios and showcases the capabilities of our LSM. Finally, we discuss the evolving DeFi landscape, considering how emerging trends, challenges, and opportunities in the blockchain ecosystem may influence the adoption and scalability of our mechanism in real-world applications.

The content of the chapter is based on [23, 24] and is organized as follows: Section 3.1 outlines our approach at a high level; Section 3.2 presents the operational semantics of our LSM mechanism and proves its formal properties in comparison to standard AMMs. In Section 3.3, we describe our netting algorithm and its computational behavior. Section 3.4 discusses our implementation and several practical application

scenarios. Finally, Section 3.5 provides a brief literature overview, explores an alternative off-line implementation of the netting mechanism, and analyzes how current trends in the blockchain ecosystem could impact the feasibility and scalability of our solution.

3.1 Approach Intuition

AMMs rely on liquidity providers who deposit tokens into AMM smart contracts to support trading activity. To reduce the cost associated with liquidity provision, many protocols implement LSMs, techniques that aim to minimize the amount of liquidity required for transaction settlement. Existing approaches in DeFi include flash loans [25, 16], flash swaps [26], and batch auctions [27]. These mechanisms allow users to borrow tokens without posting collateral, under the condition that the borrowed tokens are returned within a single transaction. However, such solutions are not well-suited for scenarios involving multiple users or trades requiring more than one atomic transaction, which limits their applicability in more sophisticated trading workflows.

A user may deposit, redeem, or swap crypto-assets, but these operations are only permitted if the user holds sufficient upfront capital to cover the transaction cost. This requirement restricts the scope of feasible interactions, particularly in scenarios where users might otherwise benefit from multi-step or cooperative strategies that temporarily exceed their available liquidity.

To introduce and analyze the mechanics of our proposed model, we present an illustrative example that demonstrates the expressive power of our mechanism. This example captures the full range of user actions and highlights how users can achieve their individual financial aims under liquidity constraints using our LSM. Consider the scenario depicted in Figure 4, involving three users A, B, and C that interact with two AMMs performing a sequence of actions $s_1 s_2 s_3 s_4 s_5 s_6$. In detail, user A owns 4 tokens of type τ_2, no tokens of types τ_0 and τ_1 (denoted with the notation A$[0 : \tau_0, 0 : \tau_1, 4 : \tau_2]$ at the top of the figure), user B has 4 tokens of type τ_0, no tokens of types τ_1 and τ_2; whereas user C owns 12 tokens

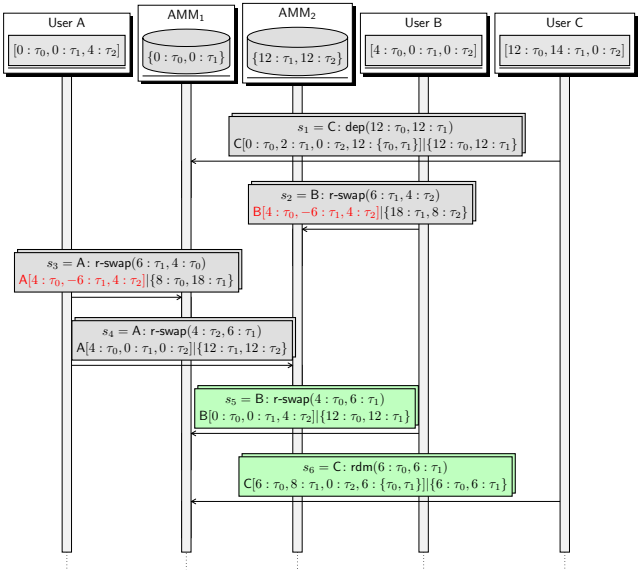

Figure 4: Interaction between three Users and two AMMs.

of type τ_0 and 14 tokens of type τ_1. The aim of users A and B is to increase their amount of token type τ_0 and τ_2, respectively. For this reason, they would like to exchange a certain amount of tokens of type τ_1 with the available AMMs with 1-to-1 rate to achieve their aims, even without having enough tokens in their wallet. Initially, user C with action s_1 creates the AMM_1 depositing 12 tokens of type τ_0 and 12 tokens of type τ_1 becoming the liquidity provider of this new AMM reserve. For this deposit, C receives equal amounts of minted tokens written as $\{\tau_0, \tau_1\}$ that work as a receipt of her deposit and that can be used afterward by C to redeem her investment. After this deposit, the AMMs have 12 tokens type of τ_0, τ_1 and τ_1, τ_2, respectively (denoted with $\{12:\tau_0, 12:\tau_1\}$ and $\{12:\tau_1, 12:\tau_2\}$ in the figure). In standard protocols like Uniswap, transactions s_2 and s_3 are rejected because users' balances cannot cover their fulfillment: B cannot swap 6 tokens of type τ_1 for 4 tokens of type τ_2, in symbols $s_2 = \mathsf{B}\colon \mathsf{r\text{-}swap}(6:\tau_1, 4:\tau_2)$, and A cannot swap 6 token of type τ_1 for 4 token of type τ_0, in symbols $s_3 = \mathsf{A}\colon \mathsf{r\text{-}swap}(6:\tau_1, 4:\tau_0)$. The

same also holds for actions s_4 and s_5 that users A and B cannot afford.

Our LSM overcomes limitations of the previous scenario: the underlying idea is to accept a momentary deficit in users' balances as long as they are covered by subsequent transactions, for instance, a swap in a different direction made by the same user or an update to the token reserve. More precisely, transactions violating liquidity constraints on user wallets are delayed and stored in a pending queue. If a subsequent transaction covers the deficit of users, all the transactions in the queue are performed atomically. Otherwise, when the queue reaches a certain length, our LSM executes a netting procedure to discard from the queue the transactions that make user balances negative, and that cannot be covered. Thus, the protocol reaches a state where no liquidity constraints are violated, and transactions can be safely performed.

Back to the example of Figure 4, our LSM allows the sequence of the actions $s_1 s_2 s_3 s_4 s_5 s_6$ (we report how each action affects the user balances and AMM reserves in the figure) to be executed since s_5 yields a state where all balances are positive, so the queue is emptied. It is not relevant if there are intermediate states where user balances are temporarily negative (colored in the figure) because they will be eventually balanced. Finally, user C redeems $6 : \{\tau_0\}$ and $6 : \{\tau_1\}$ with transaction s_6. In this way, our LSM permits a multi-party trade among A, B, and C, even when they individually do not have sufficient funds. In contrast, traditional protocols reject all four swap actions due to the lack of liquidity.

3.2 A Formal Model of the Liquidity-Saving Mechanism

This section formalizes our LSM, and proves some properties it enjoys. More precisely, we formalize user interaction and AMMs as a LTS. Our LTS builds on the work of Bartoletti [28], from which we adopt both the structure and the notation. Intuitively, LTS states represent the system configurations where we store the token supplies owned by each user and the reserves of each AMM. Note that in our model we assume that a user can submit a transaction to the AMM whenever she wants. Tran-

sitions represent transactions performed by users, which may result in an update of token supplies or AMM reserves. Throughout this model, we make the simplifying assumption that all actions are performed at no cost and that the system does not impose any transaction fees.

3.2.1 Basic definitions

We assume a set $\mathbb{T} = \mathbb{T}_0 \uplus \mathbb{T}_m$ of *tokens* (ranged over by τ, τ') defined as the disjoint union of the set of *atomic token types* $\mathbb{T}_0$ and *minted token types* $\mathbb{T}_m$. The first set includes native cryptocurrencies and application-specific tokens, whereas the second set includes tokens denoting claims that a user is acting as a liquidity provider. Minted tokens are represented by unordered pairs of distinct atomic tokens: if τ_0 and τ_1 are different atomic token types, $\{\tau_0, \tau_1\}$ is the corresponding minted token type meaning that the user has locked in an AMM a certain amount of tokens τ_0 and τ_1. In general, we denote with r, r' real numbers ($\mathbb{R}$), and, following the standard typing notation, we write $r\colon \tau$ to denote r units of (atomic or minted) tokens of type $\tau \in \mathbb{T}$.

We also assume a set of *users* $\mathbb{A}$ ranged over by A, A'. The *wallet* of a user A is represented as $\mathsf{A}[\sigma_\mathsf{A}]$, where $\sigma_\mathsf{A} \in \mathbb{T} \mapsto \mathbb{R}$ is a map storing A token supplies: $\sigma_\mathsf{A}(\tau)$ is the amount of token τ owned by A; when $\sigma_\mathsf{A}(\tau) = 0$, A owns no tokens of type τ. Given a wallet σ, a positive real number v, a token type τ, and an operation $\circ \in \{+, -\}$, we define operation $\sigma \circ v : \tau$ to increase or decrease the amount of token type τ as follows:

$$\sigma \circ v : \tau = \begin{cases} \sigma\{\tau \mapsto \sigma(\tau) \circ v\} & \text{if} \quad \tau \in \text{dom } \sigma \text{ and } \sigma(\tau) \circ v \in \mathbb{R} \\ \sigma\{\tau \mapsto v\} & \text{if} \quad \tau \notin \text{dom } \sigma \text{ and } \circ = + \end{cases}$$

where $\sigma\{\tau \mapsto v\}$ means that we update the entry of the map σ for the token type τ to the value v leaving the other entries unchanged.

We model an AMM as an unordered pair $\{r_0 : \tau_0, r_1 : \tau_1\}$, where $r_0 : \tau_0$ and $r_1 : \tau_1$ represent the reserves of tokens $\tau_0, \tau_1 \in \mathbb{T}$, with $\tau_0 \neq \tau_1$.

The *states* Γ, Γ' of the protocol are finite non-empty compositions of users', wallets and AMMs. Formally, they are defined as follows:

$$\Gamma = \mathsf{A}_0[\sigma_{\mathsf{A}_0}] \mid \cdots \mid \mathsf{A}_n[\sigma_{\mathsf{A}_n}] \mid \{r_0 : \tau_0, r_1 : \tau_1\} \mid \cdots \mid \{r_w : \tau_w, r_k : \tau_k\} \quad (3.1)$$

We assume that the operator $\mid$ is commutative and associative, and we use the notation $C \mid \Gamma$ when we want to highlight some components C (wallets and/or AMMs) of the state. In addition, we assume that initial states Γ_0 are a composition of users' and wallets as follows:

$$\Gamma_0 = \mathsf{A}_0[\sigma_{\mathsf{A}_0}] \mid \cdots \mid \mathsf{A}_n[\sigma_{\mathsf{A}_n}] \tag{3.2}$$

Moreover, for simplicity, we adopt the following assumptions for all $i \neq j$:

- each user has a single wallet ($\mathsf{A}_i \neq \mathsf{A}_j$);
- distinct AMMs cannot hold exactly the same token types ($\{r_w : \tau_w, r_{w'} : \tau_{w'}\} \neq \{r_k : \tau_k, r_{k'} : \tau_{k'}\}$).

We denote with $\Gamma_{\mathsf{A}}(\tau)$ the amount of token types τ owned by user A in the state Γ, i.e., $\sigma_{\mathsf{A}}(\tau)$ where the σ_{A} is the wallet of A in Γ. We define the *supply* of a token type τ in a state Γ, written $S_\Gamma(\tau)$, as the sum of the balances of τ in all the wallets and the AMMs in Γ. Formally, given a state Γ and a token type τ, we define $S_\Gamma(\tau)$ inductively on the structure of Γ as follows:

$$S_{\mathsf{A}[\sigma]}(\tau) = \sigma_{\mathsf{A}}(\tau)$$

$$S_{\{r_0:\tau_0,\, r_1:\tau_1\}}(\tau) = \begin{cases} r_i & \text{if } \tau = \tau_i \\ 0 & \text{otherwise} \end{cases}$$

$$S_{\Gamma \mid \Gamma'}(\tau) = S_\Gamma(\tau) + S_{\Gamma'}(\tau)$$

Given a token type τ, we denote its price in the state Γ with $P_\Gamma(\tau) \in \mathbb{R}_0^+$. For the atomic tokens, we assume that their price is given by an external oracle. The price $P_\Gamma(\{\tau_0, \tau_1\})$ of a minted token $\{\tau_0, \tau_1\}$ depends both on the supply of $\{\tau_0, \tau_1\}$ in the users' wallets and on the reserves of τ_0 and τ_1 in the corresponding AMM. More precisely, the price of a minted token $\{\tau_0, \tau_1\}$ is the ratio between the value of the corresponding AMM and its total supply in the state Γ. The value of an AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$ is obtained by summing the price of its reserves: $r_0 \cdot P(\tau_0) + r_1 \cdot P(\tau_1)$.

Therefore, we define the price of a minted token $\{\tau_0, \tau_1\}$ as follows:

$$P_\Gamma(\{\tau_0, \tau_1\}) = \frac{r_0 \cdot P(\tau_0) + r_1 \cdot P(\tau_1)}{S_\Gamma(\{\tau_0, \tau_1\})} \quad \text{if} \quad \{r_0 : \tau_0, r_1 : \tau_1\} \in \Gamma.$$

3.2.2 LSM semantics

The semantics of our LSM is defined through a two-level transition system, which provides a structured and modular approach to handling the complexities of DeFi scenarios. First, we introduce the first level, which focuses on individual actions, including deposit, redeem, and relaxed swaps. Then, we illustrate the second level that implements our LSM, coordinating the interactions between users and AMMs and managing the pending queue of actions.

First Level Semantics

Here, we concentrate on the semantics of single actions modeling the interaction between users and AMMs as transitions between states. A transition $\Gamma \xrightarrow{s} \Gamma'$ represents the evolution of the state Γ into Γ' upon the execution of the transaction s by some user. For simplicity, we consider a basic scenario in which each illustrated action is performed using the wallet $\sigma\{\tau_0, \tau_1\}$. This wallet is owned by user A; however, for notational convenience, we omit explicit reference to the user and refer only to the wallet itself.

The possible transactions that represent the labels of our LTS are:

- A: dep$(v_0 : \tau_0, v_1 : \tau_1)$ meaning that user A deposits v_0 units of token τ_0 and v_1 units of token of τ_1 in the AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$, receiving in return a certain amount of the minted token $\{\tau_0, \tau_1\}$; note that the AMM is created if it does not already exists;

- A: r-swap$(v_0 : \tau_0, v_1^* : \tau_1)$ meaning that user A transfers v_0 units of token τ_0 to AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$, possibly without having any upfront collateral, saying that she wants to receive at least v_1^* units of token τ_1 in return;

- $\mathsf{A}\colon \mathsf{rdm}(v : \tau)$ meaning a user A redeems v units of minted token $\tau = \{\tau_0, \tau_1\}$ from the AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$, receiving in return units of the atomic tokens τ_0 and τ_1.

Below, we present the semantic rules governing the state transitions for these actions.

The semantics of the deposit action $\mathsf{A}\colon \mathsf{dep}(v_0 : \tau_0, v_1 : \tau_1)$ is given by two rules depending on whether the action will result in the creation of a new AMM or in the increase of the liquidity of an already-existent AMM. In the first case, the following rule applies:

[DEPOSIT0]
$$\frac{\sigma(\tau_i) \geq v_i > 0 \ (i \in \{0,1\}) \quad S_\Gamma\{\tau_0, \tau_1\} = 0}{\mathsf{A}[\sigma] |\, \Gamma \xrightarrow{\mathsf{A}\colon \mathsf{dep}(v_0:\tau_0, v_1:\tau_1)} \mathsf{A}[\sigma - v_0 : \tau_0 - v_1 : \tau_1 + v_0 : \{\tau_0, \tau_1\}] |\, \{v_0 : \tau_0, v_1 : \tau_1\} |\, \Gamma}$$

The first premise requires user A to have the necessary amount of tokens in her wallet. The second premise $S_\Gamma\{\tau_0, \tau_1\} = 0$ implies

- that τ_0 and τ_1 are distinct atomic tokens, since otherwise $\{\tau_0, \tau_1\}$ would not be a minted token;
- that Γ is reachable and does nor contrain an AMM for the token pair τ_0, τ_1.

In return, A receives v_0 units of the minted new token type $\{\tau_0, \tau_1\}$ which user A can use to redeem her investment. Note that, as Bartoletti did in [28], we do not specify the exact number of received units because it is not critical, so any v_0 would be equally valid.

When the user provides liquidity to an existent AMM, the following rule applies:

[DEPOSIT1]
$$\frac{\sigma(\tau_i) \geq v_i > 0 \ (i \in \{0,1\}) \quad r_0 v_1 = r_1 v_0 \quad v = \dfrac{v_0}{r_0} \cdot S_\Gamma\{\tau_0, \tau_1\}}{\mathsf{A}[\sigma] |\, \{r_0 : \tau_0, r_1 : \tau_1\} |\, \Gamma \xrightarrow{\mathsf{A}\colon \mathsf{dep}(v_0:\tau_0, v_1:\tau_1)} \mathsf{A}[\sigma - v_0 : \tau_0 - v_1 : \tau_1 + v : \{\tau_0, \tau_1\}] |\, \{r_0 + v_0 : \tau_0, r_1 + v_1 : \tau_1\} |\, \Gamma}$$

The first premise requires user A to have enough funds to deposit; the second one requires that the ratio of tokens in the reserve of the AMM

is maintained; the last premise says that user A receives the amount of the minted token v that is proportional to the fraction of the provided amount of token τ_0[1] respect to the quantity in the AMM and total supply $S_\Gamma\{\tau_0, \tau_1\}$ of the minted token in the state Γ.

After the action, we reach a new state where the balances in the wallet of A and the reserves of the AMM are updated.

The semantics of a swap action $\mathsf{A}\colon \mathsf{r\text{-}swap}(v_0 : \tau_0, v_1^* : \tau_1)$ allows user A to exchange v_0 amount of token τ_0 from her wallet to the AMM and obtain back at least v_1^* amount of token τ_1. Formally, the behavior of this action is given by the following rule:

[RELAXEDSWAP]

$$\frac{v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \leq v_1^* \leq v_1}{\mathsf{A}[\sigma] \,|\, \{r_0 : \tau_0, r_1 : \tau_1\} \,|\, \Gamma \xrightarrow{\mathsf{A}\colon \mathsf{r\text{-}swap}(v_0:\tau_0, v_1^*:\tau_1)} \mathsf{A}[\sigma - v_0 : \tau_0 + v_1 : \tau_1] \,|\, \{r_0 + v_0 : \tau_0, r_1 - v_1 : \tau_1\} \,|\, \Gamma}$$

The premise of the rule requires that the actual amount v_1 of received units of τ_1, calculated based on the exchange rate in the current state, must satisfy the *constant-product invariant*, which is widely implemented in protocols such as Uniswap, SushiSwap, and Curve:

$$r_0 \cdot r_1 = (r_0 + v_0) \cdot (r_1 - v_1)$$

Moreover, compliance with this rule ensures the preservation of the reserve ratio in the AMM. If the second ($v_1 = \frac{r_1 \cdot v_0}{r_0 + v_0}$) and third premises ($0 \leq v_1^* \leq v_1$) are not satisfied, the action cannot be performed, as they are necessary conditions for respecting the invariant.

This action results in the update of A's wallet and the AMM's reserve. Note that the rule does not impose any liquidity constraints on the wallet of user A, so it can be triggered also when A owns fewer than v_0 units of tokens τ_0. In this case, the resulting wallet will be negative. Negative wallets are managed by the second level of our semantics.

[1] The actual amount of received units is irrelevant. Here we choose v_0, but any other choice would be valid.

Users can redeem units of a minted token $\{\tau_0, \tau_1\}$ for units of the atomic tokens τ_0 and τ_1. Each unit of $\{\tau_0, \tau_1\}$ can be redeemed for equal fractions of τ_0 and τ_1 remaining in the AMM. The semantics of the redeem action is given by the rule below:

[REDEEM]

$$\frac{\sigma\{\tau_0,\tau_1\} \geq v > 0 \quad v_0 = v \cdot \dfrac{r_0}{S_\Gamma\{\tau_0,\tau_1\}} \quad v_1 = v \cdot \dfrac{r_1}{S_\Gamma\{\tau_0,\tau_1\}}}{\mathsf{A}[\sigma]|\,\{r_0 : \tau_0, r_1 : \tau_1\}|\,\Gamma \xrightarrow{\mathsf{A}:\ \mathsf{rdm}(v:\tau_0,\tau_1)} \mathsf{A}[\sigma + v_0 : \tau_0 + v_1 : \tau_1 - v : \{\tau_0,\tau_1\}]|\,\{r_0 - v_0 : \tau_0, r_1 - v_1 : \tau_1\}|\,\Gamma}$$

The first premise requires user A to have the necessary amount of minted tokens in her wallet, while the other ones specify the amount of tokens to redeem from the AMM: these amounts are proportional to the quantity available in the reserve with respect to the total supply of the minted token. The resulting state is obtained by updating the wallet of A and the AMM reserves.

Finally, we introduce some notation. We write

$$\Gamma \xrightarrow{\lambda} \Gamma'$$

to denote the sequence of transactions $\lambda = s_1 \cdots s_n$ with intermediate states of the form $\Gamma_1, \ldots, \Gamma_{n-1}$ such that

$$\Gamma \xrightarrow{s_1} \Gamma_1 \xrightarrow{s_2} \cdots \xrightarrow{s_{n-1}} \Gamma_{n-1} \xrightarrow{s_n} \Gamma'$$

We say that a state Γ' is *reachable* if there exist some initial state Γ, which only contain wallets with atomic tokens and some λ such that $\Gamma \xrightarrow{\lambda} \Gamma'$. Since the transition relation defined by our semantics is deterministic, the initial state Γ together with the sequence λ uniquely determines the resulting state Γ' as well as all intermediate states. The formal definition and proof of determinism are given below in Theorem 1. Hereafter, all states considered in our results are implicitly assumed to be reachable. Moreover, we assume that when λ is the empty sequence ϵ, the execution results in an unchanged state, i.e., $\Gamma \xrightarrow{\epsilon} \Gamma$.

A *symbolic state* Γ is a state in which the amount of tokens in wallets and AMM reserves are described by arithmetic formulas over a set of

variables and numbers. A *symbolic transaction* is a transaction between symbolic states where the amount of tokens in the actions are represented by symbolic formulas rather than concrete values. We call a sequence ρ of such symbolic transactions a *pattern*.

Below, we report two key properties of our semantics that are the counterparts of the properties of Bartoletti *et al.* [28]. The first theorem states that our semantics is deterministic, and the second one says that the AMM reserves never become negative.

Theorem 1 (Deterministic semantics [28])**.** *If* $\Gamma \xrightarrow{s} \Gamma'$ *and* $\Gamma \xrightarrow{s} \Gamma''$*, then* $\Gamma' = \Gamma''$*.*

Proof. Let Γ represent the initial state and s denote the action performed by the system. We aim to prove a lemma under the assumption that s is an action of type [RELAXEDSWAP] rule. However, this lemma can similarly be proven for actions of the form [DEPOSIT] and [REDEEM] rules.

Assume by contradiction that $\Gamma \xrightarrow{s} \Gamma'$ and $\Gamma \xrightarrow{s} \Gamma''$, but Γ' and Γ'' are not equivalent. We proceed by cases on the action s. If $s = \mathsf{A}\colon \mathsf{r\text{-}swap}(v_0 : \tau_0, v_1^* : \tau_1)$ we apply the [RELAXEDSWAP] rule. Since the starting state Γ is the same for both transitions $\Gamma \xrightarrow{s} \Gamma'$ and $\Gamma \xrightarrow{s} \Gamma''$ we have that the resulting state is the same, i.e., Γ' and Γ'' are equal. This is a contradiction, so for r-swap the lemma is true. The cases for deposit and redeem actions follow similar reasoning.

□

Theorem 2 (Non depletion of AMM reserves [28])**.** *For all states* Γ*, if* $\{r_0 : \tau_0, r_1 : \tau_1\} \in \Gamma$ *then:*

1. $r_i > 0$*, for* $i \in \{0, 1\}$*;*
2. $S_\Gamma\{\tau_0, \tau_1\} > 0$*.*

Proof. For part *1.*, we proceed by induction on the length of a computation $\Gamma_0 \xrightarrow{*} \Gamma$, where Γ_0 is the initial state and $\xrightarrow{*}$ denotes the reflexive and transitive closure of the one-step transition relation $\xrightarrow{s}$.

- **Base Case:** The case with zero steps computation is trivial because Γ_0 does not contain any AMMs. In this scenario, there are no token reserves r_0 and r_1, and the property holds.

- **Inductive Case:** Assume the property holds for a computation of length n. We now show that the property is preserved when a transition $\Gamma \to \Gamma'$ is made in one more step. We note that the [DEPOSIT0] rule requires that the initial reserves of an AMM are strictly greater than zero. Furthermore, the rules that affect token reserves in AMMs ([REDEEM] and [RELAXEDSWAP] rules) have premises that ensure the reserves cannot be reduced to zero. So, the property holds for all computations of length $n+1$, as no rule can zero out the reserves.

For part *2.*, we proceed by induction on the length of the computation $\Gamma_0 \xrightarrow{*} \Gamma$, where Γ_0 is the initial state.

- **Base Case:** The case with zero steps computation is direct because Γ_0 does not contain any AMMs. In this scenario, the sum of minted tokens $S_\Gamma(\tau_0, \tau_1)$ is zero, and the property holds.
- **Inductive Case:** Assume that the property holds for a state Γ. We now show that the property is preserved under a transition $\Gamma \to \Gamma'$.

  Suppose that Γ_0 contains an AMM $\{r_0 : \tau_0, r_1 : \tau_1\}$, where $r_0 > 0$ and $r_1 > 0$, by part *1.*. We now examine each rule that infer the transition $\Gamma \to \Gamma'$:

  - **[DEPOSIT0] and [DEPOSIT1]:** These rules are trivial because deposits increase the supply of minted tokens. Thus, $S_\Gamma(\tau_0, \tau_1)$ increases, and the property holds.
  - **[RELAXEDSWAP]:** This rule is trivial because the relaxed swap actions do not affect the minted token supply. In this scenario, $S_\Gamma(\tau_0, \tau_1)$ remains unchanged, and the property holds.
  - **[REDEEM]:** Assume $\{r_0 : \tau_0, r_1 : \tau_1\} \in \Gamma$. Suppose by contradiction that the redeeming action burns all the minted tokens, i.e., it burns $v = S_\Gamma(\tau_0, \tau_1)$ units of tokens. The rule premise requires $v > 0$, which implies:

    $$r_0' = r_0 - v \cdot \frac{r_0}{S_\Gamma\{\tau_0, \tau_1\}} \quad r_1' = r_1 - v \cdot \frac{r_1}{S_\Gamma\{\tau_0, \tau_1\}}$$

    This leads to a contradiction, as it would imply $r_0' = r_1' = 0$, which contradicts the assumption that $r_0' > 0$ and $r_1' > 0$. Therefore, the minted token supply cannot be burned entirely, and $S_\Gamma(\tau_0, \tau_1) > 0$ is preserved.

□

Second Level Semantics

The second level of the transition system implements the overall mechanism of our protocol by coordinating interactions between individual actions, ensuring smooth operation and adherence to specified semantics, and upholding broader system objectives and constraints. The LTS states are called *configurations*, each defined as a 3-tuple

$$\langle \Gamma, \Gamma', \Lambda_\ell \rangle$$

where Γ is a state of the form (4.1), Γ' represents the last simulated state (as discussed below), and Λ_ℓ is a transaction queue with maximum size ℓ. In cases where the size ℓ is not central to the discussion, we use the abbreviated form Λ to simplify notation and avoid unnecessary complexity in the presentation. This queue stores the actions that result in overdrafts on users' balances that cannot be immediately executed. Formally, it is a term obtained by the following grammar:

$$\Lambda := \emptyset \mid \Lambda \circ s$$

where $\emptyset$ denotes an empty queue with no pending actions, and $\Lambda \circ s$ a queue where Λ is the head and the action s is the tail of the queue. In the semantic rules below, we use $\Lambda \circ s$ to represent when an action s is added to the queue Λ. Moreover, we write $|\Lambda|$ to indicate the size of the queue Λ. For instance, $\emptyset$ is an empty queue with no pending action; while the term $\emptyset \circ s_1$ denotes a queue with a single action s_1; and the term $\emptyset \circ s_1 \circ s_2$ denotes a queue with two actions: the head is s_1, and the tail is $\emptyset \circ s_2$. When necessary, we abuse the notation and use a queue as a sequence of transactions.

Given a state Γ we define $\langle \Gamma, \Gamma, \emptyset \rangle$ as an *initial configuration*.

The transitions of our LTS from one configuration to another have the form:

$$\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma'', \Gamma''', \Lambda' \rangle$$

and are triggered when a user submits a transaction s, and are defined by the inference rules below.

It is convenient to introduce some notation for the semantic rules. We say that a state Γ is **green** when the token supply of each user is non-negative, formally:

$$\forall\, \mathsf{A} \in \mathbb{A} \cap \Gamma, \tau \in \mathbb{T} \,.\, \sigma_{\mathsf{A}}(\tau) \geq 0$$

where we denote with $\mathbb{A} \cap \Gamma$ the set of users occurring in the configuration Γ. While we say it is **red** otherwise, namely, when

$$\exists\, \mathsf{A} \in \mathbb{A} \cap \Gamma, \tau \in \mathbb{T} \,.\, \sigma_{\mathsf{A}}(\tau) < 0$$

When a transition is triggered by an action s in a configuration $\langle \Gamma, \Gamma', \Lambda \rangle$, we have three possible scenarios.

The first scenario arises when the execution of s in Γ' (the last simulated state) using our first level semantics produces a state Γ'' that is **green**. In this case, we apply the following rule:

$$\text{[Cover]}\quad \frac{\Gamma' \xrightarrow{s} \Gamma'' \qquad \Gamma'' \textbf{green}}{\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma'', \Gamma'', \emptyset \rangle}$$

The [Cover] rule performs all the pending actions in Λ (if any), and the resulting configuration consists of the **green** state, Γ'' (in the first two components), and the emptied queue. Note that our LSM aims to maximize the number of settled actions performed by users. Moreover, our first level semantics imposes no constraints on user balances, thus, some of them could be negative in the resulting state Γ'. Differently, we maintain the checks to impose non-negativity constraints on the AMM reserve (see [RelaxedSwap] rule).

The second scenario happens when the simulation through the first level transition system of s in Γ' reaches a **red** state Γ'', and the length of Λ is less than the queue capacity ℓ (a parameter of our mechanism). In this case, we apply the following rule:

$$\text{[Overdraft]}\quad \frac{\Gamma' \xrightarrow{s} \Gamma'' \qquad \Gamma'' \textbf{red} \qquad |\Lambda_\ell| < \ell \qquad \Lambda'_\ell = \Lambda_\ell \circ s}{\langle \Gamma, \Gamma', \Lambda_\ell \rangle \xrightarrow{s} \langle \Gamma, \Gamma'', \Lambda'_\ell \rangle}$$

The [OVERDRAFT] rule enqueues s in Λ, thus, the resulting configuration consists of the current state Γ, the **red** state Γ'', and Λ' that is Λ extended with the incoming action s.

The last scenario occurs when the simulation through the first level transition system of s leads to a **red** state Γ'', and the length of Λ equals ℓ. In this case, we use the netting procedure (denoted by the function net in the following rule) to perform the *settlement* from the state Γ and the actions of Λ plus s.

The netting procedure identifies and returns a (possibly empty) sequence of feasible actions Λ'. The sequence is a subset of the input sequence Λ. The obtained subsequence Λ' is executed to obtain the resulting state Γ^*. All the other actions of Λ that are not selected by the procedure are discarded.

The resulting configuration is a triple with the state Γ^* (for the first two components) and the empty queue for the last one (the queued actions were carried out or discarded). Formally, we apply the following rule:

[NETTING]

$$\frac{\Gamma' \xrightarrow{s} \Gamma'' \quad \Gamma'' \textbf{red} \quad |\Lambda_\ell| = \ell \quad \Lambda'_\ell = net(\Gamma, \Lambda_\ell \circ s) \quad \Gamma \xrightarrow{\Lambda'_\ell} \Gamma^*}{\langle \Gamma, \Gamma', \Lambda_\ell \rangle \xrightarrow{s} \langle \Gamma^*, \Gamma^*, \emptyset \rangle}$$

It is worth remarking that the mechanism provided in this section is agnostic with respect to the netting procedure that we invoke in a black-box manner. For instance, a simple procedure could be to discard all enqueued transactions. The next section introduces a more meaningful procedure. Also, we remark that when we apply the above rule, the queue Λ contains a prefix of actions leading to a **red** state that is not balanced by other actions in the queue. Otherwise, the [COVER] rule would be applicable. The rule invokes the netting procedure to execute a subset of actions that allows the AMM to progress.

Finally, note that our configurations and semantic rules could be written without Γ' (the 2nd component of the configuration recording the simulated/speculative final state) and replacing the premise $\Gamma' \xrightarrow{s} \Gamma''$ with $\Gamma \xrightarrow{\Lambda \circ s} \Gamma''$. This approach requires re-computing the final state

every time a new action is performed. These re-computations are not efficient in an implementation where it is more convenient to store the intermediate state. We decided to follow this approach to make our formalization coherent with a possible implementation.

Theorem 3 (Deterministic semantics)**.** *If* $\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma'', \Gamma''', \Lambda' \rangle$ *and* $\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma_2, \Gamma_3, \Lambda'' \rangle$*, then* $\langle \Gamma'', \Gamma''', \Lambda' \rangle = \langle \Gamma_2, \Gamma_3, \Lambda'' \rangle$.

Proof. Assume by contradiction that $\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma'', \Gamma''', \Lambda' \rangle$ and $\langle \Gamma, \Gamma', \Lambda \rangle \xrightarrow{s} \langle \Gamma_2, \Gamma_3', \Lambda'' \rangle$, but $\langle \Gamma'', \Gamma''', \Lambda' \rangle \neq \langle \Gamma_2, \Gamma_3, \Lambda'' \rangle$. The proof proceeds by cases on the last semantic rule applied. Let's consider the case for the [Cover] rule. By the premise of the rule we have that $\Gamma' \xrightarrow{s} \Gamma''$ and Γ'' is **green** and the resulting configuration is $\langle \Gamma'', \Gamma'', \emptyset \rangle$. Moreover, by Lemma 1 we have that $\Gamma'' = \Gamma_2$, thus, $\langle \Gamma'', \Gamma'', \emptyset \rangle = \langle \Gamma_2, \Gamma_2, \emptyset \rangle$, and we have a contradiction. The cases for [Overdraft] and [Netting] rules are similar. In the case for [Netting], we also use the assumption that the algorithm net is deterministic.

□

3.2.3 Formal Properties

Our mechanism enables liquidity-saving behavior that is not possible with ordinary AMMs. An example is the simultaneous change of tokens through multiple AMMs that we presented in Section 3.1. Here, we present some theorems that characterize those scenarios where our LSM allows users to achieve some gains compared to the way used by current DeFi protocols. In Section 3.4, we will present concrete examples and specific instances corresponding to these scenarios. Those examples will clarify and instantiate our theoretical results, providing the reader with a deeper understanding of their practical implications.

In the theorems below, we use the *patterns* notion introduced above and refer to the execution process of the standard mechanism using the following transitions:

$$\Gamma \xrightarrow{s}\!\!\!\!\rightarrow \Gamma'$$

The semantics of the standard mechanism can be easily obtained from the semantics we presented in Section 3.2.2 by replacing the [RelaxedSwap]

with two rules.[2] Note that in both of the following rules, the second and third premises, namely, $v_1 = \frac{r_1 \cdot v_0}{r_0 + v_0}$ and $0 \leq v_1^* \leq v_1$, must hold in accordance with the constant-product invariant introduced earlier . If these conditions are not satisfied, the action cannot be performed.

[SWAP_1]

$$\frac{\sigma\tau_0 \geq v_0 \quad v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \leq v_1^* \leq v_1}{\mathsf{A}[\sigma] \,|\, \{r_0 : \tau_0, r_1 : \tau_1\} \,|\, \Gamma \overset{\mathsf{A}\colon \mathsf{swap}(v_0:\tau_0, v_1^*:\tau_1)}{\longrightarrow\!\!\!\!\!\rightarrow} \mathsf{A}[\sigma - v_0 : \tau_0 + v_1 : \tau_1] \,|\, \{r_0 + v_0 : \tau_0, r_1 - v_1 : \tau_1\} \,|\, \Gamma}$$

Differently from [RELAXEDSWAP] rule, the first premise checks that the wallet has enough funds, and if this is the case, update the wallet and AMM reserves accordingly. The second rule below deals with the case where the liquidity constraints are not met:

[SWAP_2]

$$\frac{\sigma\tau_0 < v_0 \quad v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \leq v_1^* \leq v_1}{\mathsf{A}[\sigma] \,|\, \{r_0 : \tau_0, r_1 : \tau_1\} \,|\, \Gamma \overset{\mathsf{A}\colon \mathsf{swap}(v_0:\tau_0, v_1^*:\tau_1)}{\longrightarrow\!\!\!\!\!\rightarrow} \mathsf{A}[\sigma] \,|\, \{r_0 : \tau_0, r_1 : \tau_1\} \,|\, \Gamma}$$

the first premise ($\sigma\tau_0 < v_0$) checks that the amount of token type τ_0 is less than the exchanged ones, so the action is ignored and the resulting state is left unchanged.

Note that, in the theorems below, we consider the pattern of symbolic actions ρ whose size is less than the maximum length of a queue ℓ, namely, $|\rho| < \ell$. Moreover, recall that we ignore the amount of transaction fees.

The following theorem characterizes those scenarios where our mechanism allows a user to successfully increase her reserve of a token type τ_0 achieving her aim, even if she initially has no tokens in her wallet. This is in contrast to the standard mechanism that, in the same scenarios, neither enables any action nor allows the user to achieve any aim.

Theorem 4 (Single user token increment)**.** *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, z : \tau_2] \,|\, \{m : \tau_0, n : \tau_1\} \,|\, \{u : \tau_2, v : \tau_1\} \,|\, \widehat{\Gamma}$$

[2]For further details on the semantics of the swap transaction, we refer the interested reader to [28, 29].

where the agent A *aims to increase the value of the token type* τ_0 *in her wallet; and where the two* AMMs *have the token type* τ_1 *in common.*

Assume that A *performs the sequence of actions* $\rho = s_1 \rho' s_2$ *of the form:*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(a : \tau_1, b : \tau_0)$ *such that* $a, b > 0$*;*
- ρ' *is a sequence of actions performed by other users, that do not cause overdraft, and that do not impact the reserves of the* AMM $\{u : \tau_2, v : \tau_1\}$*;*
- $s_2 = \mathsf{A}\colon \mathsf{r\text{-}swap}(c : \tau_2, d : \tau_1)$ *such that* $c, d > 0$, $c < z$, *and* $a \leq d$.

If $\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{\rho} \langle \Gamma', \Gamma', \emptyset \rangle$ *then* $\Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0)$*; while if* $\Gamma \xrightarrow{\rho} \!\!\!\!\rightarrow \Gamma^s$ *then* $\Gamma^s_{\mathsf{A}}(\tau_0) = \Gamma_{\mathsf{A}}(\tau_0)$.

Proof. By using our operational semantics, we have the following sequence of transitions from the initial state Γ:

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \rangle \quad \text{by [OVERDRAFT] rule} \tag{3.3}$$

$$\xrightarrow{\rho'} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ \rho' \rangle \quad \text{by a repetition of [OVERDRAFT] rule} \tag{3.4}$$

$$\xrightarrow{s_2} \langle \Gamma', \Gamma', \emptyset \rangle \quad \text{by [COVER] rule} \tag{3.5}$$

where

$$\begin{aligned}
&\Gamma''' = \mathsf{A}[b : \tau_0, -a : \tau_1, z : \tau_2] \mid \{m - b : \tau_0, n + a : \tau_1\} \mid \\
&\{u : \tau_2, v : \tau_1\} \mid \widehat{\Gamma} \\
&\Gamma'' = \mathsf{A}[b : \tau_0, -a : \tau_1, z : \tau_2] \mid \{m' : \tau_0, n' : \tau_1\} \mid \\
&\{u : \tau_2, v : \tau_1\} \mid \widehat{\Gamma'} \\
&\Gamma' = \mathsf{A}[b : \tau_0, -a + d : \tau_1, z - c : \tau_2] \mid \{m' : \tau_0, n' : \tau_1\} \mid \\
&\{u + c : \tau_2, v - d : \tau_1\} \mid \widehat{\Gamma'}
\end{aligned}$$

In the first transition (eq. 3.3), the execution of s_1 results in a **red** state so we apply the [OVERDRAFT] rule that enqueues s_1 in the pending action queue. Following this, the system evolves, taking into account the other actions ρ' that do not settle A's overdraft (so the negative supply of the user A in Γ'' of the token type τ_1 is the same as the previous configuration, so $\Gamma'_{\mathsf{A}}(\tau_1) = \Gamma''_{\mathsf{A}}(\tau_1)$), and the action in ρ' are inserted in the queue (eq. 3.4). Finally, A performs the action s_2 that covers the negative balance in her wallet, and the configuration of the system becomes **green** carrying out all the actions in the queue (eq. 3.5). Since $b > 0$, we have that $b = \Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0) = 0$, so A achieves her goal.

Whereas, when we use the standard mechanism from Γ we have that $\Gamma \overset{s_1}{\twoheadrightarrow} \Gamma$ because s_1 is discarded due to the violation of liquidity constraints; then, $\Gamma \overset{\rho'}{\twoheadrightarrow} \Gamma^s$ and $\Gamma^s \overset{s_2}{\twoheadrightarrow} \Gamma^{s'}$ because ρ' is performed by other users than A and does not affect the supply of the AMM $\{u : \tau_2, v : \tau_1\}$ and s_2 is executed affecting only τ_1 and τ_2, not τ_0 in the wallet of A. Therefore, $\Gamma^{s'}_{\mathsf{A}}(\tau_0) = \Gamma_{\mathsf{A}}(\tau_0)$: in this case user A does not achieve her goal. □

The following corollary of Theorem 4 illustrates how our mechanism enables a single user, who initially has no tokens in her wallet, to successfully increase her holdings. In contrast, the standard mechanism neither facilitates any action nor allows the achievement of any financial aim.

Corollary 4.1. *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, 0 : \tau_2] \,|\, \{m : \tau_0, n : \tau_1\} \,|\, \{u : \tau_2, v : \tau_1\} \,|\, \{p : \tau_2, q : \tau_0\}$$

where the agent A *aims to maximize the value in her wallet. Assume that she performs the sequence of actions $\rho = s_1 s_2 s_3$ of the form (where max queue size $\ell = 2$):*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(a : \tau_0, b : \tau_1)$ *such that* $a, b > 0$
- $s_2 = \mathsf{A}\colon \mathsf{r\text{-}swap}(e : \tau_1, f : \tau_2)$ *such that* $e, f > 0$
- $s_3 = \mathsf{A}\colon \mathsf{r\text{-}swap}(i : \tau_2, l : \tau_0)$ *such that* $i, l > 0$

and $l > a$ for token type τ_0, $b > e$ for τ_1, and $f > i$ for τ_2.
If $\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{\rho} \langle \Gamma', \Gamma', \emptyset \rangle$ then $\Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0)$, $\Gamma'_{\mathsf{A}}(\tau_1) > \Gamma_{\mathsf{A}}(\tau_1)$, and $\Gamma'_{\mathsf{A}}(\tau_2) > \Gamma_{\mathsf{A}}(\tau_2)$. Whereas if $\Gamma \overset{\rho}{\twoheadrightarrow} \Gamma^s$ than $\Gamma^s_{\mathsf{A}}(\tau_0) = \Gamma_{\mathsf{A}}(\tau_0)$, $\Gamma^s_{\mathsf{A}}(\tau_1) = \Gamma_{\mathsf{A}}(\tau_1)$, and $\Gamma^s_{\mathsf{A}}(\tau_2) = \Gamma_{\mathsf{A}}(\tau_2)$.

Proof. The statement directly follows by Theorem 4. Note that in this case, ρ is the action s_2 performed by the user A of the form $\mathsf{A}\colon \mathsf{r\text{-}swap}(e : \tau_1, f : \tau_2)$.

□

Finally, the following theorem characterizes those scenarios where our LSM mechanism allows the simultaneous exchanges of tokens between two users with different gains. In the lemma, we assume that the

first user aims to increase her reserve of token type τ_0, while the second user wants to increase her reserve of token type τ_2. The lemma ensures that by using our mechanism, both users can successfully achieve their respective goals. In contrast, the lemma states that the standard mechanism does not permit the users to fulfill their financial aim.

Theorem 5 (Simultaneous exchange)**.** *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, z : \tau_2] \mid \mathsf{B}[u : \tau_0, 0 : \tau_1, 0 : \tau_2] \mid \{m : \tau_0, n : \tau_1\} \mid \{p : \tau_1, r : \tau_2\}$$

where agent A *aims to increase the amount of token* τ_0 *in her wallet and* B *wants to increase the amount of token* τ_2*.*

Assume they perform the sequence of actions $\rho = s_1 s_2 s_3 s_4$ *of the form (where max queue size* $\ell = 3$*):*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(a : \tau_1, b : \tau_0)$ *such that* $a, b > 0$
- $s_2 = \mathsf{B}\colon \mathsf{r\text{-}swap}(c : \tau_1, d : \tau_2)$ *such that* $c, d > 0$
- $s_3 = \mathsf{A}\colon \mathsf{r\text{-}swap}(e : \tau_2, f : \tau_1)$ *such that* $e, f > 0$*,* $z \geq e$ *and* $f \geq a$
- $s_4 = \mathsf{B}\colon \mathsf{r\text{-}swap}(g : \tau_0, h : \tau_1)$ *such that* $g, h > 0$*,* $u \geq g$ *and* $h \geq c$

If $\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{\rho} \langle \Gamma', \Gamma', \emptyset \rangle$ *then* $\Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0)$ *and* $\Gamma'_{\mathsf{B}}(\tau_2) > \Gamma_{\mathsf{B}}(\tau_2)$*; while if* $\Gamma \xrightarrow{\rho}\!\!\!\!\rightarrow \Gamma^s$ *then* $\Gamma^s_{\mathsf{A}}(\tau_0) = \Gamma_{\mathsf{A}}(\tau_0)$ *and* $\Gamma^s_{\mathsf{B}}(\tau_2) = \Gamma_{\mathsf{B}}(\tau_2)$*.*

Proof. By using our operational semantics we have the following sequence of transition from the initial state Γ:

$$\begin{aligned}
\langle \Gamma, \Gamma, \emptyset \rangle &\xrightarrow{s_1} \langle \Gamma, \Gamma'''', \emptyset \circ s_1 \rangle \quad \text{by [OVERDRAFT] rule} && (3.6)\\
&\xrightarrow{s_2} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \circ s_2 \rangle \quad \text{by [OVERDRAFT] rule} && (3.7)\\
&\xrightarrow{s_3} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \circ s_3 \rangle \quad \text{by [OVERDRAFT] rule} && (3.8)\\
&\xrightarrow{s_4} \langle \Gamma', \Gamma', \emptyset \rangle \quad \text{by [COVER] rule} && (3.9)
\end{aligned}$$

where

$$\Gamma'''' = \mathsf{A}[b:\tau_0, -a:\tau_1, z:\tau_2] \mid \mathsf{B}[u:\tau_0, 0:\tau_1, 0:\tau_2] \mid \{m-b:\tau_0, n+a:\tau_1\} \mid \{p:\tau_1, r:\tau_2\}$$

$$\Gamma''' = \mathsf{A}[b:\tau_0, -a:\tau_1, z:\tau_2] \mid \mathsf{B}[u:\tau_0, -c:\tau_1, d:\tau_2] \mid \{m-b:\tau_0, n+a:\tau_1\} \mid \{p+c:\tau_1, r-d:\tau_2\}$$

$$\Gamma'' = \mathsf{A}[b:\tau_0, -a+f:\tau_1, z-e:\tau_2] \mid \mathsf{B}[u:\tau_0, -c:\tau_1, d:\tau_2] \mid \{m-b:\tau_0, n+a:\tau_1\} \mid \{u+c+e:\tau_2, v-d-f:\tau_1\}$$

$$\Gamma' = \mathsf{A}[b:\tau_0, -a+f:\tau_1, z-e:\tau_2] \mid \mathsf{B}[u-g:\tau_0, -c+h:\tau_1, d:\tau_2] \mid \{m+b+g:\tau_0, n-a-h:\tau_1\} \mid \{u+c:\tau_2, v-d:\tau_1\}$$

In the first transition (eq. 3.6), the execution of s_1 results in a **red** state, so we apply the [Overdraft] rule, which enqueues s_1 in the queue. Similarly, the execution of s_2 leads to a **red** state Γ''', and thus we again apply the [Overdraft] rule, enqueuing s_2 (eq. 3.7). Following this, the system performs the action s_3, which makes A's wallet balance positive without changing the overall state, which remains **red**. Whereas, the negative balance of user B in Γ'' for token type τ_1 remains unchanged, i.e., $\Gamma''''_{\mathsf{B}}(\tau_1) = \Gamma''_{\mathsf{B}}(\tau_1)$. So, the action s_3 is also inserted into the queue (eq. 3.8). Finally, B performs the action s_4 that covers her negative balance, resulting in a **green** state and executing all actions in the queue (eq. 3.9). Since $b > 0$, we have $b = \Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0) = 0$, and since $d > 0$, we have $d = \Gamma'_{\mathsf{B}}(\tau_0) > \Gamma_{\mathsf{B}}(\tau_0) = 0$, thus A and B achieve their goals.

In contrast, using the standard mechanism from Γ, we have $\Gamma \overset{s_1}{\twoheadrightarrow} \Gamma$ because s_1 is discarded due to liquidity constraint violations. Similarly, $\Gamma \overset{s_2}{\twoheadrightarrow} \Gamma$ because s_2 is discarded for the same reason. Then, $\Gamma \overset{s_3}{\twoheadrightarrow} \Gamma^s$ and $\Gamma^s \overset{s_4}{\twoheadrightarrow} \Gamma^{s'}$ because s_3 and s_4 can be executed by the standard mechanism but do not help the users achieve their goals (s_3 increases $\Gamma^s_{\mathsf{A}}(\tau_1)$ while s_4 increases $\Gamma^{s'}_{\mathsf{B}}(\tau_1)$, respectively). Therefore, $\Gamma^s_{\mathsf{A}}(\tau_0) = \Gamma_{\mathsf{A}}(\tau_0)$ and $\Gamma^{s'}_{\mathsf{B}}(\tau_2) = \Gamma_{\mathsf{B}}(\tau_2)$, meaning users A and B do not achieve their goals. □

3.3 Netting Procedure

In this section, we provide an algorithm for the netting procedure *net* invoked in the [Netting] rule of Section 3.2, and then we study some of its

properties.

3.3.1 Netting Algorithm

Our netting problem is defined as follows: given a sequence of transaction Λ and a starting state Γ, we aim to find a subsequence of transaction of Λ, call it Λ', that maximizes a given objective function f such that the execution of the transactions in Λ' from Γ lead to a **green** state Γ^*. For example, the objective function f may maximize the size of Λ', the number of involved users, or the amount of given token types, or holdings in wallets. Recall also in a **green** state the *liquidity constraints* are satisfied, namely, there is no overdraft in users' wallets. Formally, we define the following optimization problem:

$$\max\ f(\Lambda', \Gamma^*) \tag{3.10}$$

subject to

$$\Gamma \xrightarrow{\Lambda'} \Gamma^* \tag{3.11}$$

$$\Gamma^*\ \textbf{green} \tag{3.12}$$

Hereafter, we consider that the objective function f aims to maximize the size of Λ', namely, $f(\Lambda', \_) = |\Lambda'|$. Since we need an algorithm that can run on a smart contract (thus incurring affordable gas expenses), we adopt a heuristic approach that trades optimality for efficiency that avoids enumerations and attempts to maximize the number of transactions performed. To this purpose, we propose *Algorithm 1* that runs in polynomial time (quadratic in the size of the queue, at worst – see below).

Intuitively, *Algorithm 1* executes the transactions using our first level semantics of Section 3.2 that ignores the liquidity constraints. It starts by initializing Λ_r with Λ, the initial queue of pending actions, and Γ^0 with Γ, the initial state. It then starts a loop (line 2), where it simulates the execution of the actions until either the final state Γ^ℓ is not **red** or the queue Λ_r becomes empty, namely, until there are still actions to be processed and overdrafts in the system. During each iteration, we select from Λ_r the first action s_i that makes the balance of some user A negative, i.e.,

Algorithm 1 Heuristic netting Procedure Implementation

Input: $\Lambda = [s_1, s_2, \ldots, s_\ell]$, Γ
Output: Λ_r
Initialization: $\Lambda_r \leftarrow \Lambda$, $\Gamma^0 \leftarrow \Gamma$

1: $\Gamma^0 \xrightarrow{\Lambda_r} \Gamma^\ell$ ▷ Starting execution
2: **while** Γ^ℓ is **red** **and** $\Lambda_r \neq \emptyset$ **do**
3: Select $\min i$ such that $s_i \in \Lambda_r$ and $\sigma_{\mathsf{A}}(\tau) < 0$ ▷ Select action that causes overdraft
4: $\Lambda_r \leftarrow \Lambda_r - s_i$ ▷ Update actions queue
5: $\Gamma^i \leftarrow \Gamma^{i-1}$ ▷ Update last **green** state
6: $\Gamma^i \xrightarrow{s_{i+1}} \Gamma^{i+1} \xrightarrow{s_{i+2}} \cdots \xrightarrow{s_f} \Gamma^f$ ▷ Run remaining valid actions
7: $\Gamma^\ell \leftarrow \Gamma^f$ ▷ Update final state
8: **end while**
9: **return** Λ_r

the execution of s_i leads Γ^ℓ to **red** state (line 3). Then, the algorithm removes s_i from Λ_r (line 4) and updates the resulting state Γ^i (line 5) by reverting s_i. We achieve that by simply considering the previous state Γ^{i-1} that is the last **green** state. After we recover to the last **green** state, the execution is run again but from Γ^i using the actions following s_i until all balances are non-negative. Moreover, all the actions that cannot be executed due to the exchange-rate or liquidity constraints being unmet are discarded. As a result of this last execution and filtering, we obtain the **green** state Γ^f (line 6). Note that, the actions were performed from scratch, so the exchanged values may change according to modifications in the rate. These steps are iterated until we obtain a **green** state or an empty Λ_r. When this happens, the algorithm returns Λ_r. Our algorithm operates on a transaction queue that represents submissions within a specific time frame. As such, its decisions are inherently local to that window. This means that, although effective within the current snapshot, the algorithm might discard a transaction that appears to cause an overdraft now but could be covered by future incoming funds, thus missing a potentially more convenient trade over a longer time horizon.

Note that our algorithm does not strictly depend on the policy to select the action to be removed. Indeed, one could easily accommodate

new policies by modifying line 3. For example, we can replace it as follows. We first identify the user A^* and the token type τ^* with the largest overdraft in the final state Γ^ℓ, namely:

$$\tau^*, \mathsf{A}^* \leftarrow \arg\min_{\tau,\mathsf{A}} \Gamma^\ell_{\mathsf{A}}(\tau)$$

Then, we remove from the queue Λ_r the action that causes the largest overdraft of the token type τ^* in user A^*'s wallet, namely:

$$i \leftarrow \arg\min_{i} \Gamma^i_{\mathsf{A}^*}(\tau^*) \; s.t. \; s_i \in \Lambda_r \tag{3.13}$$

where $s_i = \mathsf{A}^* : \mathsf{r\text{-}swap}(\_ : \tau^*, \dots)$. In Section 3.4, Example 6 provides a concrete use case of the policy described above.

3.3.2 Netting Procedure Formal Properties

Here, we study some formal properties of our netting algorithm. First, we prove that our algorithm terminates and its complexity is quadratic on the length of the input queue in the worst-case scenario.

Theorem 6 (Termination and Complexity)**.** *Algorithm 1 always terminates, and its complexity is $O(|\Lambda|^2)$ where Λ is the input queue.*

Proof. Remember from our assumption that the length of Λ is at most ℓ. We prove termination by showing that the number of iterations of the loop is finite. In each iteration of the loop, at least one action s_i causing an overdraft is removed from Λ_r. Since Λ_r initially contains ℓ actions, in the worst-case scenario, each iteration removes at least one action, so the length of Λ decreases at each iteration. Therefore, after at most ℓ iterations, the queue Λ_r will be empty, and the algorithm will terminate.

We now consider its complexity. We know that each iteration involves finding inside the queue Λ the first action s_i causing an overdraft. This search takes $O(|\Lambda|)$ time. Removing the action s_i and re-executing the remaining actions $s_{i+1} \dots s_{|\Lambda|}$ takes $O(|\Lambda|)$ in the worst-case. Since there can be at most $O(|\Lambda|)$ iterations, the total complexity is $O(|\Lambda|^2)$, namely, the algorithm is quadratic in the length of the queue Λ. □

The following theorem guarantees that when the algorithm removes a swap action from the queue that a deposit action in the queue depends on, this last action is also removed in turn.

Theorem 7 (Dependent actions). *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, z : \tau_2] \mid \{m : \tau_0, n : \tau_1\} \mid \{u : \tau_1, v : \tau_2\}$$

Assume the user A *performs the sequence of actions* $\Lambda = p \circ s_1 \circ s_2 \circ p'$ *where* $|\Lambda| = \ell$*. The sequence of actions,* p *does not concern* τ_0 *and* τ_1*,* s_1 *and* s_2 *have the following form:*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(a : \tau_0, b : \tau_1)$ *such that* $a, b > 0$ *and* $S_{\Gamma^1}\tau_0 < 0$*;*
- $s_2 = \mathsf{A}\colon \mathsf{dep}(b : \tau_1, c : \tau_2)$ *such that* $c > 0$ *and* $S_{\Gamma^2}\tau_1 = b$*.*

and p' *is a sequence of actions that may depend on* s_2*. If* $\Gamma \xrightarrow{\Lambda} \Gamma'$ *with* Γ' ***red*** *and if* $\Gamma \xrightarrow{p} \Gamma^1$ *and* $\Gamma \xrightarrow{p \circ s_1} \Gamma^2$ *with* $\Gamma^2_{\mathsf{A}}(\tau_0) < 0$ *then* $net(\Gamma, \Lambda) = \Lambda'$ *and* s_1,$s_2 \notin \Lambda'$*.*

Proof. The hypothesis $\Gamma \xrightarrow{p} \Gamma^1$ and $\Gamma \xrightarrow{p \circ s_1} \Gamma^2$ with $\Gamma^2_{\mathsf{A}}(\tau_0) < 0$ means that s_1 does not cover any action in p but causes an overdraft concerning τ_0 for user A. This means that at a certain point during the execution of Algorithm 1, the action s_1 will be identified as the action causing the overdraft and will be removed from the queue. When Algorithm 1 reaches line 6, it identifies s_2 as a deposit that cannot be performed by user A because of lacking of funds ($\Gamma^2_{\mathsf{A}}(\tau_1) < b$), and removes s_2 as well. Therefore, the queue returned at the end of the execution of Algorithm 1 contains neither s_1 nor s_2.

□

Similarly, we can prove that, if in p' there is a redeem action of the form $\mathsf{A}\colon \mathsf{rdm}(b : \tau_1, c : \tau_2)$ (that is triggered by the execution of s_1), the Algorithm 1 discards this action too.

3.4 Implementation

In this section, we first discuss the implementation of our simulator and then present various application scenarios that highlight the advantages of our netting-based approach compared to the approach method used by standard AMM protocols.

3.4.1 Protocol Simulator

We have developed a simulator written in Haskell that implements the semantics of our LSM mechanism and our netting algorithm outlined in Sections 3.2 and 3.3. The simulator includes support for the liquidity provider transactions of deposit and redeem, as well as the r-swap transaction used to exchange tokens on the AMM. Internally, it uses the two-level semantic structure described earlier, applying either the [COVER], [OVERDRAFT], or [NETTING] rule after each transaction. It also implements Algorithm 1 with the policy to select the action to be removed described in the chapter and with the variant introduced by eq. 3.13. The simulator can be used to check the execution traces of the examples presented in the next section. It is open source and available online [22].

3.4.2 Application Scenarios

In this section, we present several examples related to application scenarios that we formally characterized in Sections 3.2.3 and 3.3.2. We ran all the following examples through our simulator to validate them.

The first example illustrates the scenario considered in Theorem 5.

Example 1 (Negative balance covered)**.** *Consider the state*

$$\Gamma = \mathsf{A}[0:\tau_0, 0:\tau_1, 0:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{8:\tau_1, 18:\tau_2\} \mid \{8:\tau_2, 18:\tau_0\}$$

Assuming a 1-to-1 exchange rate, three arbitrage opportunities arise but are only available to users with sufficient funds. Suppose agent A *wants to perform the following actions $\rho = s_1 s_2 s_3$ increasing her wallet (where max queue size $\ell = 2$):*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4:\tau_0, 6:\tau_1)$
- $s_2 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4:\tau_1, 6:\tau_2)$
- $s_3 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4:\tau_2, 6:\tau_0)$

Using the standard mechanism, all the actions are discarded because A *does not have enough balance, so the system will no evolve, namely $\Gamma \overset{\rho}{\twoheadrightarrow} \Gamma$. Since $\Gamma_{\mathsf{A}}(\tau_0), \Gamma_{\mathsf{A}}(\tau_1)$, and $\Gamma_{\mathsf{A}}(\tau_2)$ does not increase, agent* A *does not achieve her goal. Figure 5 shows a flow diagram of the actions performed adopting our*

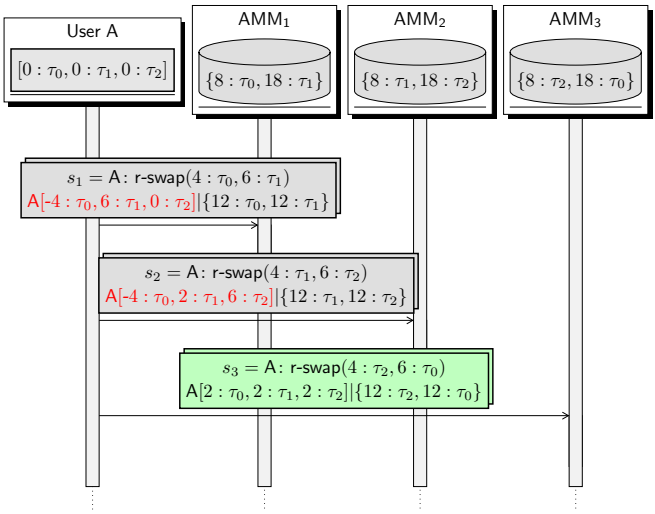

Figure 5: Flow diagram of the Example 1 among User A and three AMMs.

mechanism. By our operational semantics, we have the following sequence of transitions:

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.14}$$

$$\xrightarrow{s_2} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.15}$$

$$\xrightarrow{s_3} \langle \Gamma', \Gamma', \emptyset \rangle \quad \textit{by } [\textsc{Cover}] \textit{ rule} \tag{3.16}$$

where

$$\Gamma''' = \mathsf{A}[\text{-}4:\tau_0, 6:\tau_1, 0:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{8:\tau_2, 18:\tau_1\} \mid \{8:\tau_2, 18:\tau_0\}$$

$$\Gamma'' = \mathsf{A}[\text{-}4:\tau_0, 2:\tau_1, 6:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{12:\tau_2, 12:\tau_1\} \mid \{8:\tau_2, 18:\tau_0\}$$

$$\Gamma' = \mathsf{A}[2:\tau_0, 2:\tau_1, 2:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{12:\tau_2, 12:\tau_1\} \mid \{12:\tau_2, 12:\tau_0\}$$

The first two transitions enqueue s_1, s_2 because they cause an overdraft (eq. 3.14 *and* 3.15) *that is then covered by the last transition s_3* (eq. 3.16). *The execution of s_3 results in the* **green** *state Γ'.*

Using our mechanism the system evolves $\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{\rho} \langle \Gamma', \Gamma', \emptyset \rangle$ *reaching a state where the agent achieves her goal (increasing her wallet) since* $\Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0)$, $\Gamma'_{\mathsf{A}}(\tau_1) > \Gamma_{\mathsf{A}}(\tau_1)$, *and* $\Gamma'_{\mathsf{A}}(\tau_2) > \Gamma_{\mathsf{A}}(\tau_2)$ *(where each of them is* $2 > 0$ *for each token type).*

Note that the above scenario has similarities with the traditional finance scenario known as *gridlock* [21] where banks cannot settle their payments individually due to their insufficient liquidity. Through *netting*, each bank submits its payment instructions to designated queues, performing multilateral settlement exclusively for the net obligations. Back to our example, user A overcomes the stuck state reached by standard AMM protocols where individual swap incurs in an overdraft, enqueuing her actions and atomically performing them when reaching a positive balance.

The second example illustrates the scenario considered in Theorem 7.

Example 2 (r-swap and AMM creation)**.** *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, 6 : \tau_2] \mid \mathsf{B}[6 : \tau_0, 6 : \tau_1, 0 : \tau_2] \mid \{8 : \tau_0, 18 : \tau_1\}$$

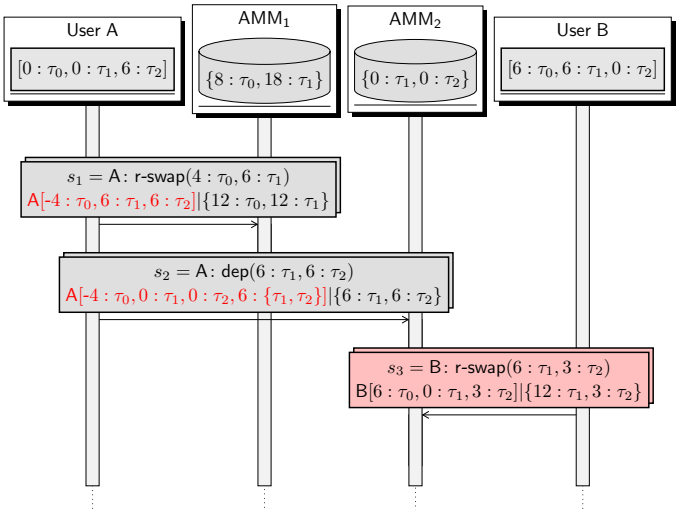

Figure 6: Flow diagram of the Lemma 7 among two Users and two AMMs.

and that agents A *and* B *want to perform the following actions* $\rho = s_1 s_2 s_3$ *(where the size of* Λ *is* $\ell = 2$*) :*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4 : \tau_0, 6 : \tau_1)$
- $s_2 = \mathsf{A}\colon \mathsf{dep}(6 : \tau_1, 6 : \tau_2)$
- $s_3 = \mathsf{B}\colon \mathsf{r\text{-}swap}(6 : \tau_1, 3 : \tau_2)$

Figure 6 shows a flow diagram of the pattern composed by a r-swap *action and a deposit action that creates the* AMM_2 *that exchanges* τ_1 *and* τ_2*.*

The evolution of the scenario is composed of the following configurations:

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.17}$$

$$\xrightarrow{s_2} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.18}$$

$$\xrightarrow{s_3} \langle \Gamma, \Gamma, \emptyset \rangle \quad \textit{by } [\textsc{Netting}] \textit{ rule} \tag{3.19}$$

where

$$\Gamma''' = \mathsf{A}[\text{-}4 : \tau_0, 6 : \tau_1, 6 : \tau_2] \mid \mathsf{B}[6 : \tau_0, 6 : \tau_1, 0 : \tau_2] \mid \{12 : \tau_0, 12 : \tau_1\}$$
$$\Gamma'' = \mathsf{A}[\text{-}4 : \tau_0, 0 : \tau_1, 0 : \tau_2, 6 : \{\tau_1, \tau_2\}] \mid \mathsf{B}[6 : \tau_0, 6 : \tau_1, 0 : \tau_2] \mid$$
$$\{18 : \tau_0, 8 : \tau_1\} \mid \{6 : \tau_2, 6 : \tau_1\}$$

The first transition results in a **red** *state, so* s_1 *enqueued in* Λ (eq. 3.17)*. Subsequently, the system evolves by considering action* s_2*, which creates a new* AMM_2 *for the token type pair* (τ_1, τ_2)*,*[3] *with* 6 *units of each token type respectively* (eq. 3.18)*. User* B *then attempts to obtain token type* τ_2 *by swapping* 6 *units of* τ_1 *on* AMM_2 (eq. 3.19)*. The action* s_2 *is added to* Λ *because the negative balance of* A *remains unresolved. Assuming that the following actions do not rectify the negative balance of token type* τ_0*, action* s_2 *saturates the pending queue* Λ*, resulting in a final configuration marked as* **red***. At this point, the netting procedure is invoked and proceeds as follows: action* s_1 *is identified as the cause of the overdraft and is subsequently discarded. According to the semantic rules governing interactions between users and* AMMs*, action* s_2 *is also discarded because* A *lacks sufficient* τ_1 *tokens to perform a* deposit *action. The removal of* s_2 *consequently leads to discarding* s_3*, as* B *depends on the* AMM *created by* s_2 *for its execution. The final configuration thus reverts to the initial state* (eq. 3.19)*. In this scenario, the actions executed by the netting mechanism are equivalent to those determined by the standard procedure.*

[3] We provide an illustrative example where a user executes a $\mathsf{deposit}_0$ action to highlight the effects of eliminating the deposit action. This example demonstrates the subsequent changes and consequences resulting from removing the deposit action.

The following two examples illustrate the scenario considered in *Lemma 5*. The latter highlights how our mechanism deviates from the standard mechanism.

Example 3 (Simultaneous Exchange)**.** *Consider a state*

$$\Gamma = \mathsf{A}[0:\tau_0, 0:\tau_1, 4:\tau_2] \mid \mathsf{B}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\}$$

and the following sequence of actions $\rho = s_1 s_2 s_3 s_4$ (where the max queue size is $\ell = 3$):

- $s_1 = \mathsf{A}\colon \textsf{r-swap}(6:\tau_1, 4:\tau_0)$
- $s_2 = \mathsf{B}\colon \textsf{r-swap}(6:\tau_1, 4:\tau_2)$
- $s_3 = \mathsf{A}\colon \textsf{r-swap}(4:\tau_2, 6:\tau_1)$
- $s_4 = \mathsf{B}\colon \textsf{r-swap}(4:\tau_0, 6:\tau_1)$

where the first and second r-swap *allow agents* A *and* B *to obtain the token type they value more, namely τ_1 and τ_2, respectively. The scenario evolves as follows:*

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma'''', \emptyset \circ s_1 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.20}$$

$$\xrightarrow{s_2} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \circ s_2 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.21}$$

$$\xrightarrow{s_3} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \circ s_3 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.22}$$

$$\xrightarrow{s_4} \langle \Gamma', \Gamma', \emptyset \rangle \quad \textit{by } [\textsc{Cover}] \textit{ rule} \tag{3.23}$$

where

$\Gamma'''' = \mathsf{A}[4:\tau_0, \text{-}6:\tau_1, 4:\tau_2] \mid \mathsf{B}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\}$

$\Gamma''' = \mathsf{A}[4:\tau_0, \text{-}6:\tau_1, 4:\tau_2] \mid \mathsf{B}[4:\tau_0, \text{-}6:\tau_1, 4:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\}$

$\Gamma'' = \mathsf{A}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \mathsf{B}[4:\tau_0, \text{-}6:\tau_1, 4:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\}$

$\Gamma' = \mathsf{A}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \mathsf{B}[0:\tau_0, 0:\tau_1, 4:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\}$

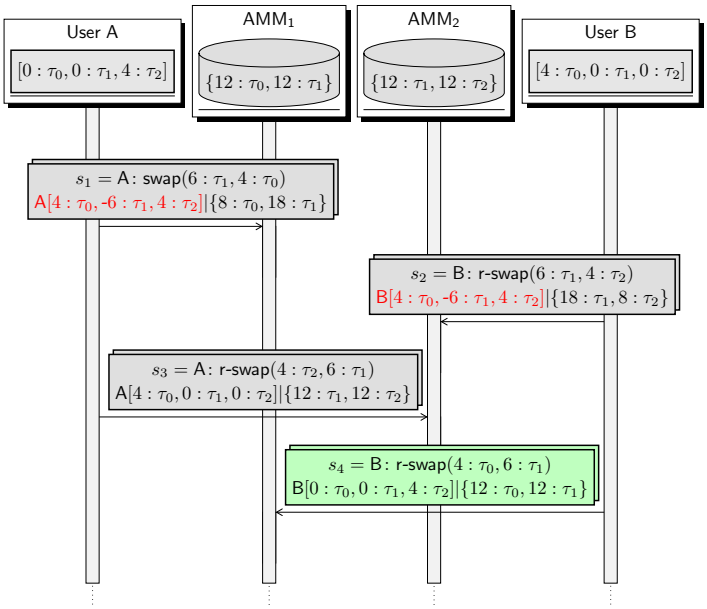

Figure 7: Interaction between two Users and two AMMs.

These swaps can generate an intermediate negative state (see $\Gamma'''', \Gamma''', \Gamma''$*) and negative net worth associated with* Users*' wallets. The execution of* s_1 *and* s_2 *is crucial to achieving the users' goal. More precisely, agent* A *wants to maximize the value of* τ_0*, while in the latter,* B *wants to maximize the value of* τ_2*. Once these* r-swap *actions are performed, the respective* AMMs *do not maintain a 1-to-1 ratio anymore. Finally, the* s_3 *and* s_4 *are performed in the other market maker to balance the previous overdraft and achieve a 1-to-1 ratio in the* AMMs *again. Once these reversed swaps are performed, the* AMMs *are not unbalanced anymore, and the users' wallets are not negative any longer.*

Analyzing each arrow of Figure 7, it is possible to visualize each relaxed swap performed in ρ*. In the context of* Γ*, both* AMM_1 *and* AMM_2 *maintain a 1-to-1 ratio among the pair token types. Simplifying the symbolic exchanges represented by the pattern* ρ *reveals that the final state of the system is equivalent to its initial state. Using our mechanism, the system evolves,* $\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{\rho} \langle \Gamma', \Gamma', \emptyset \rangle$ *where* Γ' *is a* **green** *state with both users' goal achieved:* $\Gamma'_{\mathsf{A}}(\tau_0) > \Gamma_{\mathsf{A}}(\tau_0) = 0$ *and* $\Gamma'_{\mathsf{B}}(\tau_0) > \Gamma_{\mathsf{B}}(\tau_0) = 0$*. In more detail,* User A *obtains* 4 *tokens of type* τ_0 *and* User B *obtains* 4 *tokens of type* τ_2*. Using the*

standard mechanism, the system does not evolve $\Gamma \overset{\rho}{\twoheadrightarrow} \Gamma$*, and the users do not have any increment because all the action in* ρ *violates liquidity constraints. Thus, they do not achieve their goal.*

Moreover, our LSM mechanism may settle transactions on the blockchain differently from standard AMM protocols: netting can prioritize transactions that the standard approach would discard, and vice versa. This distinction can impact user rewards, potentially encouraging the adoption of new or alternative market strategies. The following scenario exemplifies a case in which the netting algorithm could prevent swaps that would otherwise be executed using the standard protocol. The following example illustrates this difference.

Example 4 (Incomparability)**.** *Consider a state*

$$\Gamma = \mathsf{A}[0 : \tau_0, 0 : \tau_1, 4 : \tau_2] \mid \mathsf{B}[4 : \tau_0, 6 : \tau_1, 0 : \tau_2] \mid \{12 : \tau_0, 12 : \tau_1\} \mid$$

$$\{18 : \tau_1, 8 : \tau_2\}$$

Assume agents A *and* B *want to perform the sequence of actions* $\rho = s_1 s_2 s_3$ *(where max queue size is* $\ell = 2$*).*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(6 : \tau_1, 4 : \tau_0)$
- $s_2 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4 : \tau_2, 6 : \tau_1)$
- $s_3 = \mathsf{B}\colon \mathsf{r\text{-}swap}(6 : \tau_1, 4 : \tau_0)$

where the first and second r-swap *allow* A *and* B *to obtain the token type they value more, i.e.,* τ_1 *and* τ_2*, respectively. Using the standard mechanism, the evolution of the scenario is composed of the following configurations:*

$$\Gamma \overset{s_1}{\twoheadrightarrow} \Gamma \overset{s_2}{\twoheadrightarrow} \Gamma^s \overset{s_3}{\twoheadrightarrow} \Gamma^{s'}$$

The action s_1 *is discarded because* A *does not have enough balance, whereas* s_2 *and* s_3 *are performed. Thus, the system reaches the configuration:*

$$\Gamma^{s'} = \mathsf{A}[0 : \tau_0, 6 : \tau_1, 0 : \tau_2] \mid \mathsf{B}[8 : \tau_0, 0 : \tau_1, 0 : \tau_2] \mid \{8 : \tau_0, 18 : \tau_1\} \mid$$

$$\{12 : \tau_1, 12 : \tau_2\}$$

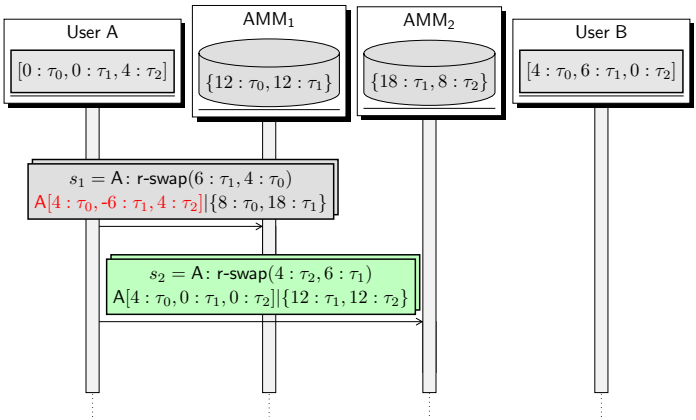

Figure 8: Flow diagram of the Example 4 among two Users and two AMMs.

Differently using our mechanism, the scenario evolves in the following configurations:

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \rangle \quad \textit{by } [\text{OVERDRAFT}] \textit{ rule} \tag{3.24}$$

$$\xrightarrow{s_2} \langle \Gamma', \Gamma', \emptyset \rangle \quad \textit{by } [\text{COVER}] \textit{ rule} \tag{3.25}$$

where

$$\Gamma'' = \mathsf{A}[4:\tau_0, -6:\tau_1, 4:\tau_2] \mid \mathsf{B}[4:\tau_0, 6:\tau_1, 0:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\}$$

$$\Gamma' = \mathsf{A}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \mathsf{B}[4:\tau_0, 6:\tau_1, 0:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\}$$

In detail, the action s_1 is enqueued (eq. 3.24)*, and when* A *performs s_2 on the second* AMM*. As the Figure 8 shows, both actions are settled because s_2 covers the overdraft on* A*'s wallet* (eq. 3.25)*. While s_3 is not executed on the first* AMM *because the execution of the previous actions changes the ratio in the reserve, and hence the swap rate. In our mechanism, executing s_2 alters the ratio in the first* AMM*, which makes s_3 no longer executable because the* [SWAP_1] *and* [SWAP_2] *rules premises $v_1 = \frac{r_1 \cdot v_0}{r_0 + v_0}$ and $0 \leq v_1^* \leq v_1$ are not met. Once $s_1 s_2$ are executed, the updated reserves allow agent* B *to swap 6 tokens of τ_1 for only 2 tokens of τ_0, which differs from the original trade she aimed to perform (exchanging 6 tokens of τ_1 for 4 tokens of τ_0). Once $s_1 s_2$ are performed,*

the ratio of the first AMM *is updated and* s_3 *is discarded because agent* B *can swap* 6 *tokens of* τ_1 *with* 2 *tokens of* τ_0 *that is a different swap amount compared to what she wants to perform (exchanging* 6 *tokens of* τ_1 *for* 4 *tokens of* τ_0*). This example highlights that the standard mechanism and ours generally settle different transactions, so they are incomparable. However, the behavior is not uncommon from what happens in ordinary* AMMs *where a user (in this case* B*) would typically decide to trade at a swap rate based on his local view or speculation on the state of* AMMs*, which can of course change if transactions of other users (in this case* A*) are executed.*

The following two examples illustrate how our netting algorithm works. The first example illustrates the behavior of Algorithm 1 as described in Section 3.3, while the second one what happens when we change the policy for selecting actions that lead to an overdraft, adopting the policy presented in eq. 3.13.

Example 5 (Netting scenario)**.** *Assume a state*

$$\Gamma = \mathsf{A}[0:\tau_0, 0:\tau_1, 4:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\} \mid$$

$$\{12:\tau_2, 12:\tau_0\}$$

and an agent A *who wants to execute the sequence of actions* $\rho = s_1 s_2 s_3$ *of the form:*

- $s_1 = \mathsf{A}\colon \mathsf{r\text{-}swap}(6:\tau_2, 4:\tau_0)$
- $s_2 = \mathsf{A}\colon \mathsf{r\text{-}swap}(6:\tau_1, 4:\tau_0)$
- $s_3 = \mathsf{A}\colon \mathsf{r\text{-}swap}(4:\tau_2, 6:\tau_1)$

Supposing that the max size of the queue is $\ell = 2$*, thus, the scenario evolves through the following configurations:*

$$\langle \Gamma, \Gamma, \emptyset \rangle \xrightarrow{s_1} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.26}$$

$$\xrightarrow{s_2} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \tag{3.27}$$

$$\xrightarrow{s_3} \langle \Gamma', \Gamma', \emptyset \rangle \quad \textit{by } [\textsc{Netting}] \textit{ rule} \tag{3.28}$$

where

$$\Gamma''' = \mathsf{A}[4:\tau_0, 0:\tau_1, -2:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\} \mid \{18:\tau_2, 8:\tau_0\}$$

$$\Gamma'' = \mathsf{A}[8:\tau_0, -6:\tau_1, -2:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\} \mid \{18:\tau_2, 8:\tau_0\}$$

$$\Gamma' = \mathsf{A}[4:\tau_0, 0:\tau_1, 0:\tau_2] \mid \{8:\tau_0, 18:\tau_1\} \mid \{12:\tau_1, 12:\tau_2\} \mid \{12:\tau_2, 12:\tau_0\}$$

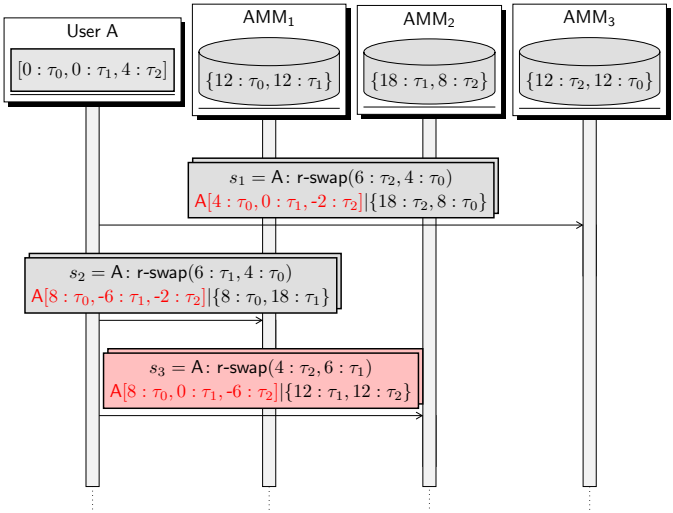

Figure 9: Flow diagram of the Example 5 among User A and three AMMs.

Upon executing the three actions illustrated in Figure 9, the system achieves a **red** *state where the user's wallet fails to meet liquidity constraints. Consequently, our mechanism triggers the* [NETTING] *rule and runs the* Algorithm 1. *During this process,* s_1 *is identified as the first action leading to an overdraft in* A*'s wallet (see line 3 of* Algorithm 1*), and is thus removed from the queue. After this adjustment, the simulation is run again, resulting in a* **green** *state, achieved through the execution of* s_2 *followed by* s_3*. It is worth noting that without* s_1*,* s_3 *covers the overdraft caused by* s_2*, thus, the reached state is* **green**.

The example above illustrates how our netting algorithm manages transactions while considering liquidity constraints: it enables the exe-

cution of actions that subsequently cover the balance overdraft, which would typically not be feasible using a standard semantics to execute the transactions.

The decision regarding the discard action now takes into account the action that causes an overdraft in the user's wallet. The simulation can re-execute the (cleaned) actions until it reaches a **green** state or the queue becomes empty. Despite the formalization and introduction of all possible actions between users and AMMs, the netting mechanism focuses on optimizing and increasing the number of r-swap actions. Moreover, even though the mechanism handles all actions traded between users and AMMs, it prioritizes the improvement of r-swap actions.

We now illustrate the flexibility of our algorithm when we adopt the policy presented in eq. 3.13.

Example 6 (Netting Maximum Overdraft)**.** *Remember that in our model the codomain of the wallet is the set of real numbers. The numerical values shown below (rounded to two decimal places) arise directly from the evolution of reserves and users' wallets during the execution of actions, as determined by the application of the transition rules.*

Consider a state

$$\Gamma = \mathsf{A}[0:\tau_0, 0:\tau_1, 4:\tau_2] \mid \{12:\tau_0, 12:\tau_1\} \mid \{18:\tau_1, 8:\tau_2\} \mid$$

$$\{12:\tau_2, 12:\tau_0\}$$

and an agent A *who wants to execute the actions* $\rho = s_1 s_2 s_3 s_4 s_5$ *of the following form (where max queue size is* $\ell = 4$*):*

- $s_1 = \mathsf{A}\colon \textsf{r-swap}(6:\tau_2, 4:\tau_0)$
- $s_2 = \mathsf{A}\colon \textsf{r-swap}(6:\tau_1, 4:\tau_0)$
- $s_3 = \mathsf{A}\colon \textsf{r-swap}(4:\tau_2, 6:\tau_1)$
- $s_4 = \mathsf{A}\colon \textsf{r-swap}(3:\tau_0, 4:\tau_2)$
- $s_5 = \mathsf{A}\colon \textsf{r-swap}(4:\tau_0, 6:\tau_1)$.

The scenario evolves according to the following configurations:

$$\begin{aligned}
\langle \Gamma, \Gamma, \emptyset \rangle &\xrightarrow{s_1} \langle \Gamma, \Gamma'''', \emptyset \circ s_1 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \\
&\xrightarrow{s_2} \langle \Gamma, \Gamma''', \emptyset \circ s_1 \circ s_2 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \\
&\xrightarrow{s_3} \langle \Gamma, \Gamma'', \emptyset \circ s_1 \circ s_2 \circ s_3 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \\
&\xrightarrow{s_4} \langle \Gamma, \Gamma', \emptyset \circ s_1 \circ s_2 \circ s_3 \circ s_4 \rangle \quad \textit{by } [\textsc{Overdraft}] \textit{ rule} \\
&\xrightarrow{s_5} \langle \Gamma', \Gamma', \emptyset \rangle \quad \textit{by } [\textsc{Netting}] \textit{ rule}
\end{aligned}$$

where

$$\begin{aligned}
&\Gamma'''' = \mathsf{A}[4 : \tau_0, 0 : \tau_1, \text{-}2 : \tau_2] \,|\{12 : \tau_0, 12 : \tau_1\} \mid \{18 : \tau_1, 8 : \tau_2\} \mid \\
&\{18 : \tau_2, 8 : \tau_0\} \\
&\Gamma''' = \mathsf{A}[8 : \tau_0, \text{-}6 : \tau_1, \text{-}2 : \tau_2] \mid \{8 : \tau_0, 18 : \tau_1\} \,|\{18 : \tau_1, 8 : \tau_2\} \mid \\
&\{18 : \tau_2, 8 : \tau_0\} \\
&\Gamma'' = \mathsf{A}[8 : \tau_0, 0 : \tau_1, \text{-}6 : \tau_2] \mid \{8 : \tau_0, 18 : \tau_1\} \,|\{12 : \tau_1, 12 : \tau_2\} \mid \\
&\{18 : \tau_2, 8 : \tau_0\} \\
&\Gamma' = \mathsf{A}[5 : \tau_0, 0 : \tau_1, \text{-}1.09 : \tau_2] \mid \{8 : \tau_0, 18 : \tau_1\} \mid \{12 : \tau_1, 12 : \tau_2\} \mid \\
&\{13.09 : \tau_2, 11 : \tau_0\} \\
&\Gamma = \mathsf{A}[1 : \tau_0, 0 : \tau_1, 2.9 : \tau_2] \,|\{12 : \tau_0, 12 : \tau_1\} \mid \{18 : \tau_1, 8 : \tau_2\} \mid \\
&\{13.09 : \tau_2, 11 : \tau_0\}
\end{aligned}$$

Upon executing action s_1, the system enters a **red** *state where the user's wallet fails to meet liquidity constraints. Once the queue is full, our mechanism triggers* [Netting] *rule and runs* Algorithm 1. *During the execution of the algorithm, s_3 is identified as the action causing the greatest overdraft in* A*'s wallet (see eq. 3.13) and is consequently removed from the queue. After this adjustment, the simulation is run again. As Figure 6 illustrates, the algorithm executes s_1, s_2, s_4, and s_5 achieving a* **green** *state.*

3.5 Related Work and Discussion

In this section, we start by exploring the related work in a broad sense. Section 3.5.1 focuses on mechanisms that have inspired our approach or addressed similar challenges. This overview highlights key solutions

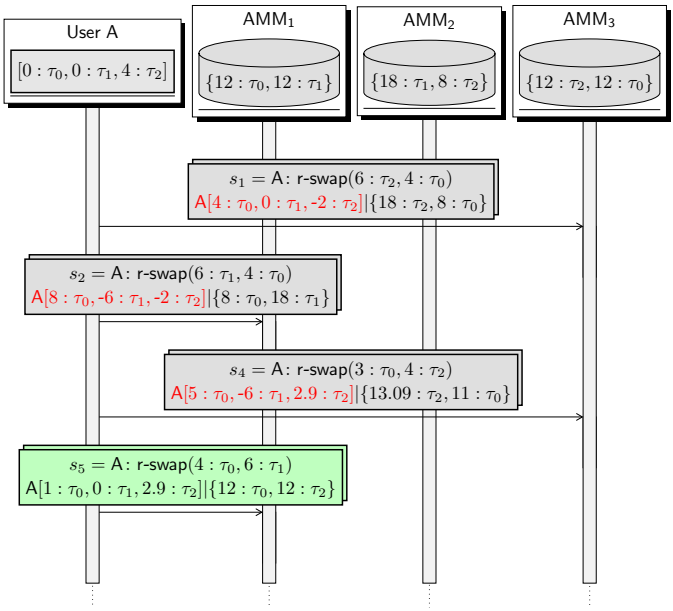

Figure 10: Flow diagram of the Example 5 among User A and three AMMs.

and frameworks from the literature that share common goals with our research or have influenced the development of our proposal. Following this, we explore an alternative design that enables offline netting in Section 3.5.2. Then, we delve into a more targeted discussion on how our mechanism can be adapted to align with the real practice of the blockchain and DeFi ecosystem. More specifically, Section 3.5.3 examines how evolving trends within the blockchain ecosystem may influence the implementation and scalability of our solution in practice.

3.5.1 Related Work

The widespread interest in distributed ledger technology has fostered the development of optimized trading protocols. This section illustrates the most relevant proposals concerning *netting mechanism*, *optimal routing problems*, and *intent-centric protocols*, and compares them with our work.

Liquidity Saving Mechanisms and Netting Approaches

Several papers have developed decentralized inter-bank payment systems where the role of the payment instructions operator is implemented through a public ledger-based protocol, and the netting mechanism through smart contracts. As a first example of this line of work, the Jasper-Ubin Project [18] investigates the possibility of achieving near-instant cross-border payments using blockchain. The project removes the single point of failure and obtains an immediate settlement without transaction reconciliation but does not give a decentralized multilateral netting. Naganuma [30] provide a secure netting protocol using the Hyperledger Fabric channel and its access control mechanism. Their secure settlement protocol does not require a specific central server and keeps part of the payment transaction information secret. Similarly, Wang [31] introduce a blockchain-based netting solution that relies on a central party and preserves the total amount of liquidity, revealing only the net amount of each bank. Cao [21] propose a decentralized netting mechanism aggregating local settlements through a smart contract to compute a globally optimal settlement. To protect privacy, they use zero-knowledge proofs to verify payment amounts confidentially. In their non-privacy-focused version, participants submit signed payment instructions with priority to the smart contract. Payments are settled immediately or queued for later netting, triggered periodically or when the queue reaches a limit. Participants calculate their nettable set and submit it with optimality proofs. The smart contract verifies and aggregates these sets until convergence or deadlock occurs, requiring additional liquidity. The protocol ensures an optimal solution if all participants act honestly. While the above papers [30, 31, 21] apply blockchain technologies to traditional financial services, our work introduces a standard financial mechanism into DeFi protocols, with the goal of implementing a liquidity-saving mechanism in the context of DeFi services. Another approach that introduces an efficient liquidity management is that proposed by Mangrove [32], a decentralized exchange protocol based on the concept of *reactive liquidity*. The main idea is to allow users to post "offers" (smart contracts) with-

out locking funds. Offers are then performed by so-called "takers", who try to execute the offers, obtaining the needed funds if available. In that way, liquidity is only committed when needed. Uncovered transactions in our approach share some similarities in spirit with Mangrove offers, in the sense that they are enqueued without locking funds. However, both approaches differ in the way liquidity saving is achieved: roughly, Mangrove relies on a sort of smart contract call-back approach, while in our case we use netting.

Optimal Routing Problems

Another relevant research field investigates how to execute several trades of different crypto-assets on networks of multiple CFMMs. These approaches are known as *routing problems* on DEXs. Angeris[33] and Diamandis[34] pointed out that the optimal routing problem can be formulated as an efficiently solvable optimization problem. Solving such a problem means determining the most efficient path for a trade, i.e., a sequence of crypto-assets exchanges in a network of CFMMs that realizes a given trade, to maximize the user's utility and minimize its costs.

Danos [35] introduce arbitrage scenarios within exchange networks and develop an effective global routing system. In detail, they explore how optimal routing strategies are employed to capitalize on price differentials across multiple exchanges, enhancing opportunities for profitable arbitrage.

Similar to those approaches, our mechanism aims at finding a solution to maximize a specific parameter. However, the main difference between ours and the above-mentioned papers [33, 34, 35] is the objective function to maximize. As illustrated in Section 3.3, our objective function consists of maximizing the number of actions in the queue, i.e., the number of transactions to settle. On the other hand, those proposals formulate the optimization problem in terms of the largest possible user's utility, and their routing problem tries to detect the most convenient and profitable route for executing trades across AMMs of the same and different token types belonging to the network.

Intent-Centric Protocols

A significant area of research in Ethereum ecosystem focuses on addressing the aggregated token swap problem, i.e., determining a sequence of swaps on AMMs that realizes a given trade by improving the liquidity of users. Various DeFi applications, such as UniswapX [36], Uniswap V4 [37], and CoW Protocol [27], tackle this challenge by adopting *intent-centric protocols*. These protocols shift the focus from transaction execution to defining users' desired outcomes, creating competitive routing marketplaces, introducing gas-free cross-chain swaps, and incorporating batch auctions to discover profitable prices.

UniswapX implements *intent-centric swaps*, combining on-chain and off-chain liquidity for a competitive trading marketplace. Uniswap V4 enhances liquidity with *hooks*, enabling gas-free cross-chain swaps and offering flexibility via customizable features. CoW Protocol utilizes batch auctions for efficient price discovery, and CoW Swap, its decentralized exchange interface, introduces CoW Hooks that allow executing custom actions before and after trades. While existing protocols seek to optimize trading across multiple AMMs, our approach introduces a distinctive LSM designed to maximize users' ability to execute trades they define autonomously. We formalize this through the notion of a financial aim, representing a user-specified objective alongside the precise sequence of actions required to achieve it.

Although the notion of intent employed in previous protocols differs from our implementation of financial aims, primarily in that the former delegates execution strategies to the protocol, both approaches can be viewed as alternative implementations of action aggregation.

3.5.2 Discussion on Off-line Netting

One of the key characteristics of our mechanism is that netting is delegated to users. This assumes that users will have implicit incentives to achieve netting either by submitting transactions that cover overdrafts (cf. rule [Cover]) or by filling up the queue and triggering the netting procedure (cf. rule [Netting]). Besides the absence of explicit incentives, there

is an additional issue. Netting via [NETTING] rule can be very expensive (e.g., in terms of gas) for the user filling up the queue and triggering the netting algorithm. In practice, a user will only fill up the queue if she knows that the netting procedure will result in enough benefit for her, to compensate for the gas cost. Since the netting procedure proposed would be openly available and is (polynomially) efficient, it is not unrealistic to expect that users can predict the outcome of the netting procedure and decide to trigger it if it makes sense for them. Still, it is natural to wonder whether we could provide an alternative design to our mechanism, where netting happens offline, and incentives to achieve netting are *explicit* and part of the blueprint of the mechanism. Below, we sketch how our model can be extended to accommodate more realistic environments.

Explicit Netting via User-submitted Netting Proposals. A first feature that we could add to our mechanism is an explicit netting operation that allows users to submit netting proposals (i.e., subsequences of the current queue) at any moment. The rule [OFFLINENETTING] allows a user to submit, at any time, a *netting proposal* consisting of a subsequence $\Lambda' \subseteq \Lambda$ of the current queue. The proposal is accepted if executing Λ' from the current intermediate state Γ' leads to a state Γ''**green** satisfying the netting condition. When this is the case, the system atomically executes the proposed subsequence, clears the queue, and updates the global state to Γ''. Intuitively, this models an off-chain computation of a netting solution that is later verified and finalized on-chain. Formally speaking, by introducing a new transaction $net(\Lambda')$, this could be formalized as follows:

$$\text{[OFFLINENETTING]}\quad \frac{\Lambda' \subseteq \Lambda \quad \Gamma' \xrightarrow{\Lambda'} \Gamma'' \quad \Gamma''\textbf{green}}{< \Gamma, \Gamma', \Lambda > \xrightarrow{\mathsf{A}:\, net(\Lambda')} < \Gamma'', \Gamma'', \emptyset >}$$

The [OFFLINENETTING] rule could be enriched with additional constraints, for example, to enforce that the submitted proposal Λ' is of a certain size (e.g., non-empty, a proportion of the current queue, etc.). However, the above proposed solution would still leave incentives to perform netting

implicit in the selected transactions, which may lead to undesired behaviors. For instance, a group of users may be interested in netting their own transactions, discarding the transactions of the rest of the users. To remedy this, one could consider incentive mechanisms that explicitly associate some value gain to the transactions in the queue. We discuss below two alternative approaches.

Explicit Netting Incentives via Transaction-based Rewards. One possible approach to achieve reward-based incentives would be as follows:

- Introducing the *reward factor* $\epsilon \in \mathbb{R}_{]0;1[}$ to specify a fraction of the tokens involved in the transaction, to be offered as a reward for including the transaction in the queue.
- Adding the *reward wallet* $\Omega : \mathbb{T} \to \mathbb{R}_+$ to the contract, which is increased by withholding an ϵ fraction from every transaction, and paid out to the user performing the netting operation. To denote this reward in a specific state Γ we write: $\Omega_\Gamma(\tau)$.
- Enriching the state of the contract so that now a state Γ also consists of the current reward Ω, in addition to AMMs and users. Formally states would be of the form:

$$\Gamma = \Omega \mid \mathsf{A}_0[\sigma_{\mathsf{A}_0}] \mid \cdots \mid \mathsf{A}_n[\sigma_{\mathsf{A}_n}] \mid \{r_0 : \tau_0, r_1 : \tau_1\} \mid \cdots \mid \{r_w : \tau_w, r_k : \tau_k\}$$

The rule [RELAXEDSWAPWITHREWARD] extends the standard relaxed swap by introducing an explicit reward mechanism. A fraction ϵ of the output amount v_1 is withheld from the user and accumulated in the reward wallet Ω, while the remaining $(1 - \epsilon) \cdot v_1$ is credited to the user's wallet. The reward wallet thus aggregates contributions from all transactions in the queue and represents the total compensation available to the user who later performs a successful netting operation. This mechanism ties rewards directly to transaction volume and ensures that netting incentives scale with the economic activity in the queue. To formalize how rewards are collected rule [RELAXEDSWAP] would be amended as follows,

where changes w.r.t. the original rule are highlighted in **red**:

$$\frac{v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \le v_1^* \le v_1 \cdot (\mathbf{1} - \epsilon)}{\begin{array}{c} \mathsf{A}[\sigma] \,|\, \{r_0 : \tau_0, r_1 : \tau_1\} \,|\, \boldsymbol{\Omega} \,|\, \Gamma \xrightarrow{\mathsf{A}:\ \mathsf{r\text{-}swap}(v_0:\tau_0, v_1^*:\tau_1)} \\ \mathsf{A}[\sigma - v_0 : \tau_0 + (\mathbf{1} - \epsilon) \cdot v_1 : \tau_1] \,|\, \{r_0 + v_0 : \tau_0, r_1 - v_1 : \tau_1\} \\ |\, \boldsymbol{\Omega} + \epsilon \cdot \mathbf{v_1} \,|\, \Gamma \end{array}}$$

[RELAXEDSWAPWITHREWARD]

Other AMM transactions, such as deposits and redeems, could be modified similarly.

The rule [OFFLINENETTINGWITHREWARD] formalizes user-submitted netting in the presence of explicit transaction-based rewards. A user A proposes a subset of pending transactions $\Lambda' \subseteq \Lambda$ which, when executed from the current intermediate state Γ', produces a state Γ'' satisfying the netting condition **green**. Upon successful netting, the user A collects the rewards accumulated for the transactions included in Λ', stored in the reward environment $\Omega_{\Gamma''}$. These rewards are transferred to the balance of A, after which the reward environment is reset, the transaction queue is cleared, and the system state is updated accordingly. To make the reward reset explicit, we introduce the initial reward function Ω_{Γ_0}, defined by:

$$\Omega_{\Gamma_0} : dom(\Omega) \to \mathbb{R}_{\ge 0} \qquad \text{with} \qquad \forall \tau \in dom(\Omega).\ \Omega_{\Gamma_0}(\tau) = 0$$

Only the rewards associated with transactions included in Λ' are collected; the rewards corresponding to excluded transactions are forfeited, thereby incentivizing users to include as many, or as valuable, transactions as possible in their netting proposals.

$$\frac{\begin{array}{c} \Lambda' \subseteq \Lambda \quad \Gamma' \xrightarrow{\Lambda'} \Gamma'' \quad \Gamma'' \mathbf{green} \quad \mathbf{\Gamma''} = \mathbf{A}[\sigma] \,|\, \mathbf{\Omega_{\Gamma''}} \,|\, \mathbf{\Gamma'''} \\ \mathbf{\Gamma^*} = \mathbf{A}\Big[\sigma + \sum_{\tau_i \in dom\ \boldsymbol{\Omega}} \cdot \mathbf{\Omega_{\Gamma''}}(\tau_i) \ :\ \tau_i\Big] \,|\, \mathbf{\Omega_{\Gamma_0}} \,|\, \mathbf{\Gamma'''} \end{array}}{< \Gamma, \Gamma', \Lambda > \xrightarrow{\mathsf{A}:\ net(\Lambda')} < \mathbf{\Gamma^*}, \mathbf{\Gamma^*}, \emptyset >}$$

[OFFLINENETTINGWITHREWARD]

It is worth noticing that the collected reward is $\Omega_{\Gamma''}$, i.e., it only comprises the reward associated with the transactions that have been included in

netting; the reward of excluded transactions is lost. Therefore, users willing to perform netting transactions are incentivized to include as many transactions as possible (or rather, the highest value of transactions as possible).

Explicit Netting Incentives via MEV. An alternative way to incentivize users to submit netting proposals with rewards directly associated with transactions in the queue would be to allow them to extract MEV [38] (see below). This can be achieved, for instance, by resorting to strategies for extracting optimal MEV from AMMs such as the Dagwood sandwiches of [39].

One way to achieve this would be to allow users to submit a supersequence on the current queue, allowing them to insert their own transactions at any place while still requiring netting to result in a **green** state. Thus, the rule [OFFLINENETTINGMEV] captures an alternative incentive model based on MEV extraction. Here, a user A may submit a supersequence $\Lambda' \supseteq \Lambda$ of the current queue, allowing her to insert additional transactions of her own while preserving the original queued transactions. The proposal is accepted only if executing Λ' results in a **green** state. The restriction that all additional transactions in $\Lambda' \setminus \Lambda$ must belong to A prevents unauthorized manipulation of other users' actions. By allowing strategic insertion of transactions, this rule enables A to extract MEV as compensation for performing netting, while still enforcing global netting correctness. The rule for user-submitted netting transactions is formalized as follows:

$$\frac{\Lambda' \supseteq \Lambda \quad \Gamma' \xrightarrow{\Lambda'} \Gamma'' \quad \Gamma'' \quad \forall \mathbf{s} \in (\mathbf{\Lambda'} \setminus \mathbf{\Lambda}).\mathbf{s} = \mathsf{A}\colon \mathsf{f}(\dots)}{< \Gamma, \Gamma', \Lambda > \xrightarrow{\mathsf{A}\colon net(\Lambda')} < \Gamma'', \Gamma'', \emptyset >} \text{[OFFLINENETTINGMEV]}$$

The premise $\forall \mathsf{s} \in (\mathbf{\Lambda'} \setminus \mathbf{\Lambda}).\mathsf{s} = \mathsf{A}\colon \mathsf{f}(\dots)$ ensures that A is not inserting transactions by other users. The premise $\Lambda' \supseteq \Lambda$ forbids A to drop transactions from the queue. This means that this kind of netting operation would only work for those cases where the first transaction is an overdraft by A, who is then incentivized to recover. Said premise could be

relaxed to allow A to discard overdrafts by other users, or transactions that reduce MEV strategies (e.g. redeem operations [39]).

3.5.3 LSM Adaptation to DeFi Trends

Decentralized environments inherently present challenges such as inconsistent views of the system state and information asymmetry among participants. These conditions can affect settlement processes and may create opportunities for certain actors to exploit differences in information access or timing. In particular, issues like transaction reordering, selective inclusion or exclusion, and price slippage are well-documented in the context of decentralized applications and remain the subject of ongoing research. For example, Lührs [40] developed encrypted mempools to address the MEV issues. Although our primary focus is on liquidity management in DeFi protocols, we acknowledge the relevance of these challenges also in our setting. We do not address them here, but we consider them important directions for future investigation.

More specifically, MEV [13] refers to the maximum value that miners, validators, or block producers can extract from transaction ordering, inclusion, or exclusion within a block. MEV typically arises in DeFi systems, where the ability to reorder transactions can lead to profits through arbitrage, front-running, or liquidation opportunities.

While MEV can improve market efficiency by correcting price inconsistencies, it can also harm users by creating opportunities for slippage, front-running, and unfair market manipulation. In Ethereum, MEV has been traditionally earned by miners in the Proof-of-Work (PoW) model, but after the transition to Proof-of-Stake (PoS), validators are now those who extract MEV.

Flashbots [41] is a service designed to optimize the extraction of MEV while maximizing rewards for validators. It establishes a private transaction pool, where non-mining participants (searchers) submit immutable bundles of transactions. These bundles are then relayed to participating validators, ensuring the transactions are executed efficiently and profitably, bypassing the public Ethereum mempool. Flashbots introduces a

proposer-builder separation[4], wherein specialized block builders optimize blocks for validators to maximize MEV extraction. According to a 2023 report [42], about 95% of Ethereum transactions are routed through Flashbots, with validators earning over $100,000$ ETH in additional rewards, significantly increasing their median block reward.

Flashbots plays a significant role in the Ethereum MEV ecosystem, with more than 70% of validators using its MEV-Boost[5] service. Though MEV transactions account for less than 1% of all Ethereum transactions, Flashbots handles 60-80% of MEV-related transactions, dominating the market.

In this setting, while the liquidity-saved AMM might function correctly, Flashbots could capture all the netting rewards, reducing user benefits. For instance, user-proposed netting may not work for regular users, as Flashbots could steal or replay netting solutions, even though this outcome might still be acceptable if the primary goal of efficient netting is achieved.

This service computes the optimal block for miners or validators from user-submitted transactions, extracting MEV and providing miners with a share of these rewards through higher block fees and direct payments. Our mechanism automates the submission of transactions, reducing unnecessary trades and increasing efficiency. Once the netting mechanism is executed, the resulting transactions are sent to validators.

However, routing these transactions through Flashbots introduces several challenges. Flashbots can act as an MEV adversary by controlling transaction flow, potentially capturing all netting rewards and reducing user benefits from optimized trading. It may extract rewards from user transaction optimizations, front-run or reorder netting proposals, and *manipulate transaction* queues by controlling their order, inclusion, or exclusion, rendering them ineffective or even *stealing netting proposals* for its own gain. Another issue involves *commit-reveal schemes*. It could hide transaction details initially to prevent front-running but may still be

[4] https://ethresear.ch/t/proposer-block-builder-separation-friendly-fee-market-designs/9725

[5] https://docs.flashbots.net/flashbots-mev-boost/introduction

vulnerable to censorship by Flashbots. If the reveal transaction is censored and fails to make it into a block, the commit becomes ineffective, negating the intended protections. Thus, while Flashbots enhances block profitability for validators and addresses some MEV inefficiencies, it also introduces concerns related to transaction control and value capture that must be considered when designing MEV-related mechanisms.

Chapter 4

Intent-based Protocols

DEXs platforms have facilitated billions of dollars in on-chain transactions [43], demonstrating the viability of DeFi markets. However, compared to CeFi, DeFi still faces critical challenges, particularly in terms of capital efficiency, execution quality, and user experience.

Indeed, traditional DeFi protocols require users to specify exact execution details for transactions (e.g., asset pairs, paths, and slippage tolerances), thus presenting limitations such as user complexity, fragmented liquidity, inefficient execution, and vulnerability to MEV exploitation [29].

To address these limitations, intent-based protocols introduce solver-mediated architectures where decentralized competition among solvers turns high-level user goals, known as intents, into on-chain execution, combining on-chain and off-chain liquidity to fulfill them, offering a fundamentally different execution paradigm from that of traditional DEXs. In protocols like CoW [27], UniswapX [36], and Anoma [44], users submit intents – such as token swaps or complex multi-party transactions – that are executed by external agents, competing to find optimal outcomes. This paradigm simplifies user interactions and optimizes transaction costs because these agents, to fulfill user requests, can aggregate liquidity, sequence transactions, and execute them across multiple DEXs. In contrast, traditional aggregators (like 1inch [45] and Matcha [46]) focus on optimizing routing across multiple on-chain DEXs to provide im-

mediate execution. However, these systems remain confined to public on-chain liquidity and are inherently susceptible to MEV attacks due to their reliance on transparent, mempool-based execution.

The separation between intents and transaction execution enables intent-based protocols to optimize outcomes, reduce on-chain complexity, and enhance cost efficiency and user experience. By delegating execution to specialized agents, these protocols also foster more competitive pricing and, with honest solvers, provide stronger protection against adversarial behavior. Solvers privately aggregate, route, and batch these intents before submitting them for on-chain settlement, limiting mempool visibility and exploitable metadata. Batch auctions settle many intents at a uniform clearing price, eliminate time-priority races, make common MEV strategies (e.g., front-running, sandwiching) far less profitable, and represent a significant step toward stronger MEV mitigation. Moreover, by enabling multiple agents to compete in fulfilling intents, they create a more flexible and user-centric financial infrastructure, better aligned with the fragmented and diverse nature of modern blockchain ecosystems. Intent-based protocols enable more flexible coordination, enhanced privacy, and potentially fairer settlement via mechanisms such as batch auctions and peer-to-peer matching. Although these protocols are gaining attention from developers and DeFi users, little is yet known about their fundamental characteristics and underlying dynamics.

To fill this gap, this chapter first presents a systematic review of the relevant literature and an empirical analysis of several real-world protocols, and then provides a formal model of the core mechanisms underlying intent-based protocols in terms of a labeled transition system (LTS). The model captures the key components of these protocols (including intent aggregation, solver competition, and trade settlement) and precisely characterizes the interactions with users and external agents. Our model serves as a formal foundation for analyzing and verifying protocol-level properties such as fairness, scalability, and MEV resistance [47, 48, 49], while also providing a flexible design framework for building more efficient, composable, and user-aligned DeFi protocols. While prior approaches often rely on heuristics or focus narrowly on routing efficiency,

our contribution offers a unified and extensible semantic model that generalizes intent-based execution across diverse DeFi architectures. By abstracting execution logic from low-level details and integrating heterogeneous settlement (e.g., batch auctions, RFQ, market makers), our model supports the principled analysis and guides the practical design of intent-based financial systems.

Our work investigates how intent-based protocols work and provides the formal basis for the development of more efficient, secure, and user-centric DeFi infrastructures. The chapter is organized as follows: Section 4.1 introduces the relevant background on intent-based protocols. Section 4.2 provides a systematization of knowledge of the relevant literature and an empirical analysis of several real-world protocols. Section 4.3 presents our formal model, focusing on the inference rules governing state transitions. Section 4.4 exemplifies our model through an illustrative scenario. Section 4.5 compares our work with the relevant literature and positions our paper within the broader context of intent-based protocol design, details the abstractions and limitations of our model, and outlines promising directions for future research.

4.1 Context and Motivation

Blockchain interoperability, the capacity of independent networks to exchange data and coordinate state, has become essential to overcome today's fragmentation, where assets and information remain confined to their native chains. Specialized protocols and middleware, such as bridges, cross-chain messengers, and standards frameworks, enable cross-chain connectivity, yet users still have to manage route selection, fees, timing, and trust trade-offs. The resulting push for deeper interoperability has driven a steady evolution in how blockchain operations are executed. In DeFi, practice has moved from the *standard swap*, where the user chooses the venue and path, sets slippage, gas, and deadlines, and signs and submits the transaction, to the *atomic cross-chain swap*, which enforces settlement across independent chains.
Figure 11 illustrates a swap flow in which *Alice selects Uniswap on Ethereum,*

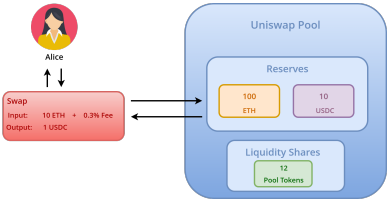

Figure 11: Uniswap *swap* mechanism.

sets 0.5% *slippage and a 5-minute deadline, then swaps* 10 *ETH for* 1 *USDC*, gaining fine-grained control at the cost of complexity and MEV/front-running exposure. In standard swaps, users pay gas for each trade and the transaction typically executes almost immediately at the quoted rate or very close to it. As a cross-chain example in Figure 12, consider the scenario from [50]: *Carol sells her Cadillac for Bitcoins; Alice buys the Cadillac using an Ether; Bob trades Ether for Bitcoins; they arrange a swap in which Alice sends Ether to Bob, Bob sends Bitcoins to Carol, and Carol transfers the blockchain-recorded title to Alice, with atomicity guaranteeing safety across chains*. Even in this case, participants must orchestrate counterparties, timeouts, and scripts.

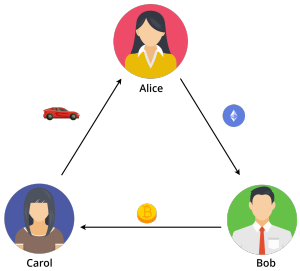

Figure 12: *Cross-chain swap* mechanism.

In contrast to conventional DeFi, where users must select transaction details and paths, set slippage, fees, and timing, and submit transactions manually, intent-based protocols deliver a safer and more scalable experience while preserving on-chain verifiability at settlement and lifting user interaction from imperative transaction building (*how*) to declara-

tive goal specification (*what*). A user states the desired outcome of the form: " *I want* 100 *token B and am willing to pay up to* 100 *token A*", and the system determines the execution path. Conceptually, intents are composable and portable instructions: they describe target state transitions without prescribing routes; multiple intents can be aggregated and composed by solvers into valid transactions (batching and multi-venue routing); and they can be propagated across decentralized networks via a gossip layer, enabling fault-tolerant, peer-to-peer dissemination without centralized coordinators. This abstraction reduces user cognitive load, increases flexibility, and separates their expression from on-chain settlement, borrowing ideas from intent-driven networking and autonomous service management. The macro effect is liquidity consolidation: without intents, users access one bridge at a time; with intents, competing solvers aggregate liquidity across chains, bringing a larger effective pool to the user. Competition among solvers, via auctions, Request For Quote windows, or solver races, tends to improve prices and latency under the user's constraints.

4.2 Intent-based Protocol Analysis

In this section, we examine the core elements of intent-based protocol design and implementation. Firstly, Section 4.2.1 provides an in-depth analysis of the intent abstraction and its alignment with the Ethereum standardization. Section 4.2.2 then analyzes the role of *solvers* and the execution mechanisms deployed in current protocols. After that, Section 4.2.3 discusses multi-chain execution and the interoperability techniques adopted by intent-based systems. Building on this conceptual background, Section 4.2.4 analyzes the major platforms and compares them along the main phases of a generic intent-based workflow. This comparative study highlights the design choices that matter in each stage and defines the requirements we incorporate into our formal model, guiding a general specification that abstracts across concrete implementations.

4.2.1 Intents

Intents express multi-step financial workflows as a single, declarative, and flexible request, representing a clear advance over traditional DeFi transactions. Although the notion of intent has been formalized by Anoma, the core principles of intent-based systems have emerged over time through mechanisms such as limit orders (enabling conditional execution), solver-mediated auctions, gas sponsorship (decoupling fee payment from transaction origination), delegation systems for access control, and transaction batching or DEX aggregators to improve execution quality and pricing efficiency. These innovations set the foundation for contemporary intent-centric architectures. What follows are examples of protocols that implement this architecture in practice. The CoW Protocol directly embodies the Coincidence of Wants (CoW) principle, allowing users to submit intents that can be matched peer-to-peer or cleared via batch auctions. UniswapX introduces an off-chain intent layer, where users post orders that are fulfilled by competing solvers who source liquidity from any available venue. 1inch Fusion builds on the Request-for-Quote (RFQ) model (see the following section), where users request price quotes from market makers who commit to fulfilling them, adding batch auction logic to enhance execution efficiency and solver competition. Hashflow [51] also adopts an RFQ-like structure, enabling users to submit structured quote intents that market makers fulfill with pricing commitments. Finally, Anoma exemplifies a fully generalized intent-centric architecture, enabling users to broadcast high-level intents that solvers fulfill by assembling atomic, multi-party transactions, advancing the flexibility and modularity of decentralized execution.

Ecosystem Intent Standards

Intent-based systems are increasingly the preferred interface for user cross-chain activity, abstracting the complexity and timing constraints of traditional bridges. The expanding multi-chain landscape makes it difficult to maintain liquidity and solver coverage, often increasing costs, latency, and failure rates. To address this problem, standardization is emerging

at the ecosystem layer: shared formats and interfaces let projects interoperate and reuse infrastructure (order dissemination, settlement, and solver networks), intensifying competition to fulfill user intents and improving outcomes. Designs differ, but they converge on intent-centric interoperability across protocols, chains, and applications: *ERC-7683* is the most practical choice for EVM developers today, *ERC-7521* generalizes intents for smart accounts, and the *SUAVE* intent format explores privacy-preserving, cross-domain execution within auction pipelines. Establishing a shared, censorship-resistant, low-latency, and verifiable intent layer that is built on open standards and paired with neutral, open solver-selection mechanisms, would deliver composability, fairness, and best-execution across Ethereum and the wider multichain ecosystem.

ERC-7683 (Cross-Chain Intents). ERC-7683 [52] is a draft for a standard for cross-chain intents proposed by Uniswap Labs and Across and aims to provide a framework that simplifies and standardizes cross-chain actions. In ERC-7683 architectures, cross-chain execution is organized around the following roles:

- **Swapper:** the user who signs an off-chain message encoding a `CrossChainOrder` that specifies the settlement contract, deadlines, the origin chain, and any implementation-specific `orderData`.

Listing 4.1: `CrossChainOrder` signed by the swapper.

```
struct CrossChainOrder {
    address settlementContract;  // Contract that settles
        the order
    address swapper;             // User initiating the
        swap
    uint256 nonce;               // Replay protection
    uint32  originChainId;       // Origin chain ID
    uint32  initiateDeadline;    // Latest time to open on
        origin
    uint32  fillDeadline;        // Latest time to fill on
        destination
    bytes   orderData;           // e.g., tokens, amounts,
        destination chains, fees
}
```

- **Filler:** the executor that initiates the transaction on the origin chain, performs the fill on the destination chain(s), and claims rewards; implementations can tailor price resolution, constraints, and settlement logic.

Listing 4.2: Resolved view for filler planning and accounting.

```
struct ResolvedCrossChainOrder {
    address settlementContract;
    address swapper;
    uint256 nonce;
    uint32  originChainId;
    uint32  initiateDeadline;
    uint32  fillDeadline;
    Input[]  swapperInputs;   // Assets taken from the
        swapper
    Output[] swapperOutputs;  // Assets delivered to the
        swapper
    Output[] fillerOutputs;   // Assets delivered to the
        filler as reward
}

struct Input {
    address token;    // ERC20 token address
    uint256 amount;   // Token amount
}

struct Output {
    address token;    // ERC20 token (address(0) = native)
    uint256 amount;   // Token amount
    address recipient;// Recipient address
    uint32  chainId;  // Destination chain ID
}
```

- **Settlement Contract:** the interface exposes minimal entry points (e.g., `initiate` and `resolve`) to open and materialize orders. For precise execution planning, a `ResolvedCrossChainOrder` view expands the order into explicit inputs/outputs so both swapper and filler know the exact tokens, amounts, and chains involved.

Listing 4.3: Minimal settlement contract interface.

```
interface ISettlementContract {
    function initiate(
        CrossChainOrder calldata order,
        bytes calldata signature,
```

```
            bytes calldata fillerData
        ) external;

        function resolve(
            CrossChainOrder calldata order,
            bytes calldata fillerData
        ) external view returns (ResolvedCrossChainOrder memory
            );
    }
```

ERC-7521 (Generalized Intents for Smart Accounts). ERC-7521 [53] is a standard supporting generalized intents for smart contract wallets. The aim of the standard is to create a framework for smart contract wallets to integrate with and automatically support a wide range of possibilities defined by the signed intents themselves. ERC-7521 enables atomic multi-intent execution: multiple intents can be bundled into a single solution that finalizes only if every intent's conditions are met. If any intent fails, the entire transaction (including already processed steps) reverts, preserving fairness and state integrity.

Listing 4.4: ERC-7521 `UserIntent` struct.

```
struct UserIntent {
    address sender;    // smart account making the intent
    bytes[] intentData; // intent segments; first 32 bytes = standard ID
    bytes signature;  // wallet-provided signature data
}
```

The `UserIntent`, illustrated in Listing 4.4, encapsulates a user's action or operation request and consists of:

- `address sender`: This field identifies the originator of the intent, allowing the system to authenticate and authorize the request.
- `bytes[] intentData`: This array contains the actual data of the intent, divided into segments. Each segment represents a specific part of the user's overall intent and is processed individually.
- `bytes signature`: This is a digital signature that provides security and integrity, ensuring the intent has not been tampered with and is indeed from the claimed sender.

Listing 4.5: ERC-7521 `IntentSolution` struct.

```
struct IntentSolution {
    uint256      timestamp; // evaluation time
    UserIntent[] intents;   // intents to execute
    uint256[]    order;     // execution order
}
```

While the `IntentSolution`, shown in Listing 4.5 and in the Listing example 4.6, is a structure that groups multiple `UserIntent` objects together for processing and is composed of:

- `uint256 timestamp`: A timestamp marking the creation or submission time of the solution.
- `UserIntent[] intents`: An array of `UserIntent` objects that are part of the solution.
- `uint256[] order`: This array specifies the order in which the segments of different intents should be executed.

In ERC-7521, each `UserIntent`'s `intentData` is split into ordered segments that execute sequentially within the intent, while segments across multiple intents can be interleaved according to a shared `IntentSolution.order`. An entry point contract validates the sender and signatures, then routes each segment to the correct intent standard contract by reading the first 32 bytes (the `standardId`) and using its internal mapping `standardId` → `contract`. The destination standard contract interprets and executes the segment's semantics. In Figure 13[1], the intent processing order like `[2,1,2]` runs (Intent 2, seg1) → (Intent 1, seg1) → (Intent 2, seg2).

[1]The figure is inspired by the online article [54].

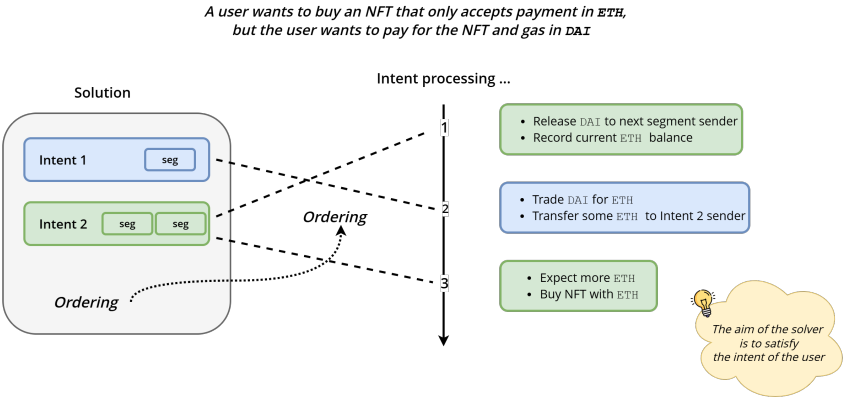

Figure 13: An example of intent processing ordering.

Segments within an intent's `intentData` execute in their declaration order. Across multiple intents, however, segments may be interleaved according to the ordering specified by the `IntentSolution` (see Listing 4.6).

Listing 4.6: Example of `IntentSolution`.

```
IntentSolution{
 timestamp : 12342354
 intents : [
      UserIntent{'0xsolver' ,['seg1_data'], '0xsolver_singature'
          } ,
      UserIntent{'0xuser', ['seg1_data', 'seg2_data'], '0
          xuser_signature'}
     ]
 order : [2,1,2]
}
```

SUAVE Intent Format. Flashbots' SUAVE[2] defines a chain-agnostic intent representation that captures user preferences and constraints independently of the execution venue. By routing intents through a single auction layer, SUAVE orchestrates block construction and private execution while safeguarding privacy and curbing MEV. A shared schema

[2]SUAVE stands for Single Unifying Auction for Value Expression.

allows wallets and dApps to target a single interface, enables solver networks to compete in intents-based MEV auctions, and helps builders assemble trust-minimized blocks from cross-domain order flow. The result is reduced fragmentation, greater portability of intents, and a foundation for interoperable, privacy-preserving block construction across the ecosystem. In SUAVE, users express intents by authoring EVM programs that declare objectives and permissible operations, including allowlists of contracts permitted to access confidential user data; practical templates can support non-expert users.

Listing 4.7: Illustrative example of a SUAVE intent and corresponding bid.

```
{
  "core": {
    "accounts": [{ "address": "0xUser...", "chainId": 8453 }],
    "id": "intent-001"
  },
  "constraints": {
    "permittedChains": [8453, 42161],
    "deadline": 1730505600,
    "maxGas": "5000000000000000",
    "slippagePercentage": 1.0,
    "dispensableAssets": [{ "chainId": 8453, "address": "0xUSDC
        ..." }],
    "desiredAssets":     [{ "chainId": 42161, "address": "0xWETH
        ..." }]
  },
  "goal": {
    "type": "swap",
    "sellToken": "0xUSDC...",
    "buyToken":  "0xWETH...",
    "sellAmount": "1000000000",
    "minBuyAmount": "530000000000000000"
  }
}
```

Listing 4.8: SUAVE solver bid with executable step.

```
{
  "bidId": "0xabc123...",
  "steps": [
    {
      "sequenceNo": 1,
      "chainId": 8453,
      "validUntil": 1730507400,
      "solution": {
```

```
        "to": "0xRouter...",
        "calldata": "0xabcdef...",
        "value": "0"
      }
    }
  ]
}
```

4.2.2 Solvers

Intent execution is carried out by third-party agents – such as solvers in CoW Protocol, fillers in UniswapX, resolvers in 1inch Fusion, or relayers in Across – which we collectively called *solvers* hereafter. They are decentralized agents that monitor available intents, solve constraint-satisfaction problems to match compatible orders and route across venues, and construct transactions that satisfy application-specific validity predicates such as slippage bounds, deadlines, and authorization checks. Each of them determines and executes an optimal strategy to fulfill user intents and internalize routing mechanisms. They improve execution quality and risk management: they mitigate adversarial microstructure effects (including sandwich attacks), provide atomicity guarantees where available, and assume interim risks such as inventory, pre-funding, and latency. The competition among solvers provides optimal fulfillment strategies, fosters a dynamic and market-driven execution environment, and enables a flexible protocol design, improving usability. This mechanism enhances execution quality and mitigates adversarial risks. Intents can avoid public mempool visibility (e.g., via RFQs, private relays, or batch auctions), limit leaked metadata (prices, paths, timing), and commit to orders before revealing details, making front-running/sandwiching far harder. Solver algorithms can optimize for multiple objectives such as minimizing gas costs, maximizing liquidity utilization, and preserving privacy. This paradigm builds on techniques from DEX routing, which split and sequence trades across venues to obtain the best all-in price, and generalizes them to arbitrary state transitions and cross-domain workflows, enabling coordinated execution across chains, markets, and applications. Taking the auction mechanism into account, solver behavior is

shaped by two cost components: entry and effort/congestion. Because auction entry requires fixed investments in connectivity, monitoring, and pre-positioning inventory, a solver will enter only if the expected surplus from winning outweighs the entry cost. Once in, the solver chooses an effort level and incurs a variable cost that typically grows with competition; greater effort can improve executable prices (through better routing and aggregation) and increase the probability of winning. Solvers spend resources before selection – probing venues, locking quotes, staging inventory, and paying on-chain simulation or transaction fees – which are non-recoverable if their bid fails. These pre-win costs feed back into optimal entry and effort choices and thus determine equilibrium participation and intensity.

Auction mechanisms

Trade execution in DeFi typically follows one of two paradigms: *continuous* trading or *auction-based* settlement. Continuous trading is the default in AMMs, where users exchange one asset for another against pooled reserves. These protocols process transactions sequentially on a first-come, first-served basis; earlier orders can influence later ones through price impact, but not vice versa. By contrast, intent-based protocols allocate liquidity and choose execution plans through *auctions* where solvers compete and all orders compete simultaneously and symmetrically, enabling collective price discovery and a single clearing price that aggregates demand and supply. These mechanisms govern which transactions are selected, how they're ordered, and when they're included in blocks – shaping the blockchain's efficiency, fairness, and economic incentives. The auction-based settlement and, in particular, the competition among solvers incentivize efficient routing, align solver rewards with user value, and provide a scalable foundation for user-centric execution.

Intent-based systems admit multiple auction styles, each shaping who competes, what is revealed, and when execution occurs. Starting from the simplest *no-selection* flow, where intents are stored in a mempool and solvers race to fill them first. *RFQ markets* move competition off-chain,

where users solicit quotes, compare all-inclusive prices, and accept a signed offer for on-chain settlement. Some approaches curate *private intent pools*, inviting a limited set of solvers to bid, while *public intent pools* open the same process to anyone. *Dutch auctions* start with a high executable price that decays over time until a solver accepts, aligning speed with willingness to earn less. And when coordination is crucial, *batch auctions* collect many intents and clear them together, trading a bit of latency for improved price discovery, lower slippage, and fewer time-priority games.

Table 3[3] summarizes typical benefits, drawbacks, and representative examples for each auction type.

Table 3: Auction mechanism in intent-based protocols.

Auction type	Benefits	Drawbacks	Examples
Private Intent Pools	Decouples user–solver relationship; customizable selection method.	Potential for centralized control; operator downtime risk; adds latency compared to other methods.	Across
Public Intent Pools	Public verifiability; shared intent order flow.	Difficult to combine competing protocols' order flows; adoption challenges.	Anoma; SUAVE
Request for Quote (RFQ)	Decentralized and simple to implement; efficient pricing; low latency.	Relies on browser stability; connectivity issues due to weak P2P support; creates liveness dependency.	Uniswap; Hashflow
Dutch Auction	Potentially lower fees over time; encourages competitive bidding; quick price discovery; MEV protection; speedy settlement, predictable decay schedule.	High initial fees may deter users; low initial fees may deter solvers; single-asset focus; fragmented liquidity;inventory-dependent/centralization pressure.	UniswapX; 1inch Fusion
Batch Auction	Increases efficiency by handling multiple intents simultaneously; leverages aggregated demand.	Complexity in managing grouped transactions; potential delays due to batching.	CoWSwap

Below, we provide detailed descriptions of the three mechanisms most commonly used in DeFi intent-protocol implementations – RFQ, Dutch, and batch auctions.

[3]The table is inspired by the online article [6].

Request For Quotes (RFQ). RFQ is a trading mechanism where a user specifies her intent expressing the asset, order side (buy/sell), and size, then requests prices from different solvers. It is commonly used in traditional finance and is increasingly used for flexible liquidity in multi-chain markets. Authorized solvers compete to satisfy it, aggregating liquidity that may be on-chain or off-chain while minimizing information leakage until execution. The user can accept one of the quote solvers' proposals or decline. The competition among solvers ensures that users receive the most favorable prices for their transactions. RFQ is widely used in over-the-counter (OTC) markets[4] and across several crypto venues, with settlement executed bilaterally or through an exchange/venue that intermediates the RFQ process.

Dutch auction. A Dutch auction [56] is a trading mechanism where the seller sets a high initial price that decreases at fixed intervals or continuously until a buyer accepts it. The order creator specifies the auction's duration and the maximum price the seller accepts for the transfer (start price), and the lowest one (end price). Once the auction begins, as illustrated in Figure 14, the price decreases linearly from the start price to the end price over the time interval; if no bid is received by expiration, the auction ends and the order returns to the seller. This mechanism enables rapid, transparent price discovery and swift settlement, but auctioning tokens individually can fragment liquidity.

There are different types of Dutch auctions: the *descending-price auction*, the *ascending-price auction*, and the *sealed-bid auction*. In a *descending-price auction*, the price of the item starts high and gradually decreases until a buyer is found. In an *ascending-price auction*, the price of the item starts low and gradually increases until a buyer is found. In a *sealed-bid action*, all seller submit their bids simultaneously, and no participant knows the bids of others. The winner and payment are determined when the bids are opened. Dutch auctions reduce effective fees as prices decay:

[4] *"An OTC market is a decentralized market where participants trade securities not listed on formal exchanges. OTC trading is mainly facilitated by broker-dealer networks and lacks the strict regulations of centralized national exchanges."*[55]

Figure 14: In Dutch auction, price decreases over time, bids placed above the current price are accepted in the order received, and the auction ends once the reserve price is reached.

because participants can wait on a public, deterministic curve rather than fight gas wars, failed attempts, and overbids decline. The shared descending schedule also sharpens solver competition, with all parties racing to lock in the best remaining terms. Because users sign off-chain and execution follows a published timetable rather than live AMM quotes, they limit front-running and sandwich attacks, delivering quick price discovery and predictable, time-based settlement.

Batch auctions. A batch auction is a trading mechanism that aggregates user intents into batches over time windows and auctions them off to competing solvers that find the most optimal settlement path. This competition transforms batch auctions into a form of order flow auction, designed to determine the optimal prices for user trades. While Dutch auctions handle single-token trades efficiently, batch auctions share liquidity across orders and support simultaneous multi-asset settlement. Solvers search – on-chain AMMs, private/off-chain liquidity, and peer-to-peer netting across the batch – to propose settlement plans. The winning plan maximizes total price surplus for the batch and is executed on-chain as a single atomic transaction, powering complex cross-asset

workflows, eliminating coordination overhead, and preserving uniform per-asset clearing prices. Batch auctions directly address MEV: all orders for the same token pair within a batch clear at a uniform price, and the atomic execution makes transaction reordering ineffective. By grouping trades and enforcing uniform settlement, batch auctions reduce slippage, improve fairness, and protect users against exploitative behaviors. Moving toward near block-sized batches preserves these protections while keeping delays minimal, approaching the speed of continuous execution. The main negative aspect of batch auctions is latency, as batches must remain open briefly to collect orders, introducing latency; however, tightening windows brings performance close to block-time settlement.

4.2.3 Multi-chain Structure

The blockchain landscape, especially the DeFi ecosystem, has become irreducibly multi-chain: value and applications are dispersed across independent networks that do not natively interoperate, straining liquidity, degrading user experience, and limiting capital efficiency. To address the ensuing challenges, multi-chain networks aim to provide decentralized infrastructure that connects different blockchains and enables communication between them. In this setting, a multi-consensus model is often employed: different, interconnected chains run distinct consensus mechanisms, allowing the network to leverage each algorithm's strengths. The objective is to satisfy diverse requirements, such as scalability, security, and transaction finality, while preserving the flexibility needed to compose applications across chains. *Interoperability* is the capacity for different blockchain networks to communicate, exchange data, and coordinate state across differing protocols, consensus mechanisms, and execution models. It enables assets, information, and contract calls to move across chains, allowing users to deploy capital on one network, invoke services on another, and settle transactions where latency and fees are lowest. We distinguish two broad categories of interoperability: *asset*-level and *data*-level. Our analysis focuses on asset-level interoperability, which directly supports the model we propose. Asset interoperability refers to the

transfer and exchange of assets among multiple parties. An asset is a digital representation of value in various forms, including goods, services, currencies, and financial instruments such as stocks and bonds. On the blockchain, assets are primarily represented as tokens, which can be securely stored and traded using cryptographic keys. As Li *et al.*[57] illustrate, asset interoperability is typically realized through three patterns: *(i) lock/burn–mint*, which transfers value by locking or burning assets on the source chain and minting corresponding value on the target chain; *(ii) atomic cross-chain swaps*, which convert assets across chains without a trusted third party; and *(iii) liquidity-based exchange*, which enables users to swap native assets across chains via liquidity pools across different chains. We next examine cross-chain swaps, outlining the structure of cross-chain exchange engines and their practical implementations.

Cross-chain Swaps

Cross-chain connectors enable messaging across blockchains by creating shared hubs between networks. In practice, these hubs, such as bridges, routers, and gateways, standardize message formats, reliably relay data across networks, and let independent chains communicate. They provide the foundational layer for genuine blockchain interoperability. Building on that foundation, cross-chain swaps enable users to trade assets across different blockchains without relying on a centralized intermediary, thereby combining crypto bridging with token swaps. They are a primitive for multi-chain DeFi because they: *(i)* eliminate the need to bridge and then swap in separate steps, streamlining asset movement; *(ii)* collapse complex, multi-hop transfers into a single transaction; and *(iii)* help optimize liquidity by coordinating tokens across chains without maintaining balances in different wallets. As Figure 15 illustrates, cross-chain bridges are protocols that allow tokens and data to move between different blockchains. They operate by either locking assets on the source blockchain and issuing equivalent tokens on the destination blockchain or by burning tokens on the source chain and reminting them on the destination chain. For instance, a cross-chain bridge can lock Ethereum (ETH) on the Ethereum blockchain and mint Wrapped

Ethereum (WETH) on the Binance Smart Chain (BSC), allowing the asset to be utilized within the BSC ecosystem. While bridges and swaps are closely related, they differ in scope: a cross-chain bridge usually transfers one token for its equivalent on another chain, whereas a cross-chain swap transfers funds and facilitates their exchange for another asset on the destination chain. For example, a user might bridge ETH from Ethereum to Solana and then immediately swap it for a Solana-native token through a cross-chain DEX.

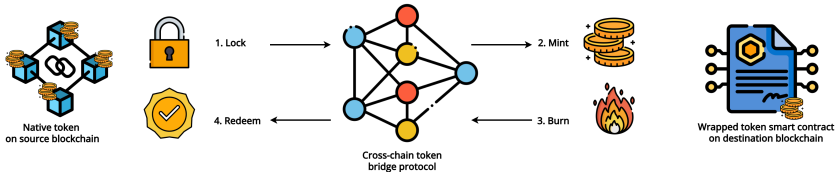

Figure 15: Execution flow of the lock/mint token bridges mechanism.

Cross-chain bridges are powered by three main mechanism types. As

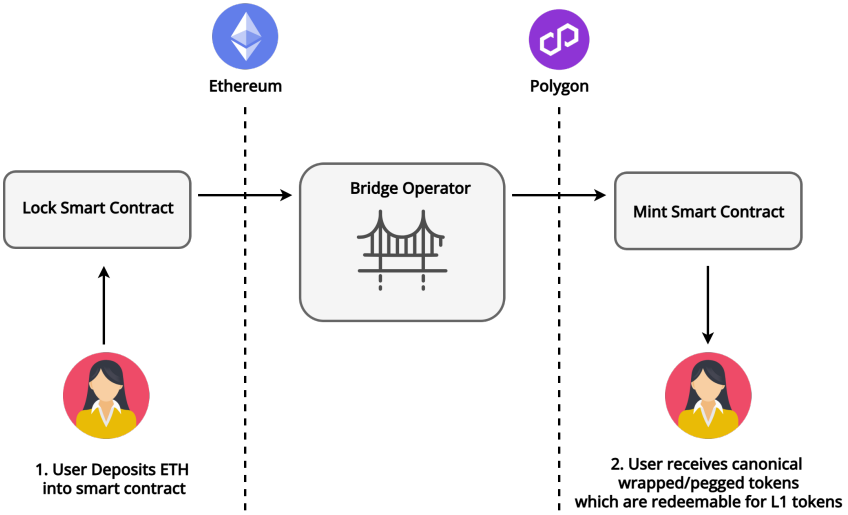

Figure 16: Lock/mint bridge mechanism.

Figure 16 illustrates, in the *lock/burn–mint* model, a user locks tokens in

a smart contract on the source chain, and wrapped versions are minted on the destination chain. In the *burn/mint* model, tokens are burned on the source chain and re-issued natively on the destination chain, conserving global supply without wrapped assets. In the *lock/unlock* model, tokens are locked on the source chain and matched by unlocking native tokens from a liquidity pool on the destination chain, typically incentivized through liquidity provision. Beyond token transfers, bridges can also facilitate arbitrary data messaging, enabling the execution of cross-chain contract calls once tokens are transferred.

Despite their potential, cross-chain mechanisms are prone to failure that may occur on the source chain (e.g., broadcast errors, insufficient gas, or inadequate token approvals), on the destination chain (e.g., DEX contract errors, blocklisted tokens, or violations of minimum/maximum transfer limits), or within the bridge itself (e.g., insufficient liquidity, pool imbalances, or timeouts due to congestion). Successful execution depends not only on correct on-chain logic but also on the robustness of the bridge infrastructure.

Beyond operational failures, cross-chain swaps are particularly vulnerable to MEV that arises when validators, or bots manipulate transaction ordering and timing to extract profit. The multi-step approach of cross-chain swaps exposes a larger attack surface: sandwich attacks in mempools can exploit slippage; bridge relayers may reorder or delay transactions for arbitrage; settlement lags create opportunities for timing arbitrage; and even intent-based protocols concentrate MEV extraction in the hands of sophisticated solvers. As a result, ensuring the safety of cross-chain swaps requires enforcing atomicity and consistency across chains designing mechanisms to mitigate adversarial extraction of value.

An expanding set of cross-chain swap protocols provides various pathways to interoperability, scalability, and security, reflecting the diverse architectures that drive today's multi-chain DeFi ecosystem. For example, Chainlink [58] focuses on decentralized oracle services and its Cross-Chain Interoperability Protocol (CCIP), which delivers reliable data and executes transactions across chains, while Polygon [59] enhances Ethereum's scalability through sidechains, enabling faster and cheaper transactions

while remaining compatible with the Ethereum ecosystem.

4.2.4 Intent-Based DEX Platforms on Ethereum

DEX aggregators represent cross-chain connectors, price-taking routers that query many liquidity venues, like AMMs, compute one or more paths (splitting the trade), and submit a single on-chain transaction on behalf of the user. They are deterministic mechanisms that, once given quotes at a specific time, immediately settle them without relying on competitive solver networks. Aggregators perform liquidity discovery by routing submitted trades across venues to minimize slippage, raise fill probability, and reduce fees. By selecting optimal paths, they improve execution quality and overall cost efficiency. Intent-based DEX aggregators represent an evolution in cryptocurrency trading, offering enhanced user experiences, MEV protection, and cross-chain functionality that traditional DEXs cannot match. They operate differently from other DEX aggregators because, instead of manually executing trades on AMMs, users sign trading intentions that specify desired outcomes rather than executing specific transactions, allowing for optimized execution through competitive solver networks.

Table 4 summarises the top Ethereum DEX aggregators by 30-day revenue (DeFiLlama). We then provide an overview of the emerging intent-based DeFi landscape protocols. Although their architectures differ, intents are emerging in the Defi ecosystem.

Table 4: Ethereum execution layers: DEX Aggregators and Intent-based DEXs.

#	Protocols	Class	Execution model	Notable feature(s)
1	**1inch Fusion+**	Intent-based DEX & RFQ	Pathfinder routing; public API/SDKs	
2	**0x Aggregator (Matcha)**	DEX aggregator	Router & RFQ	Swap API; pro MM RFQ alongside AMMs
3	**KyberSwap Aggregator**	DEX aggregator	Router & split routes	100+ integrated DEXs; dynamic routing
4	**Velora**	DEX aggregator & RFQ	Delta-intents engine; multi-agent execution	
5	**ODOS**	DEX aggragator	Smart order routing	Path splitting; route visualization
6	**CoW Protocol**	Intent-based DEX	Off-chain Batch auctions; FCA	Uniform clearing; MEV-aware settlement
7	**UniswapX**	Intent-based DEX	Off-chain Dutch auction & RFQ	Gasless fills; MEV protection; external fillers
8	**Hashflow**	RFQ DEX	Off-chain MM RFQ, on-chain settle	Zero-slippage claims; MEV-resilient quotes

1inch Fusion+. 1inch Fusion+ [45] is an intent-based DEX aggregator where resolvers compete in *Dutch auctions* to execute gasless and MEV-protected swaps. Users submit limit orders with a variable exchange rate encoded as a time-decaying price curve with four parameters: the auction start timestamp (the time signature plus a short waiting period to avoid premature starts), an auction start rate (the highest executable rate at or before auction start), a minimum return amount (the floor below which the order cannot be filled), and a decrease rate (the value that defines how the exchange rate decays over time). With 1inch Fusion Dutch auctions, the exchange rate for an asset swap order starts high and gradually decays from the desired rate toward the minimum until a resolver accepts and executes limit-orders. Every swap has a specific time window during which a resolver can execute it. Resolvers monitor the time-decaying price curve, execute once the quote crosses their individual profitability threshold, and split orders into 6-10 parts, creating multiple fill points that improve achieved rates while preserving resolver economics. If a resolver does not take the trade before its validity window closes, it should be resubmitted. The price curve is dynamic and adapts to gas market conditions, managing gas price volatility, reducing the chance of order expiration, and speeding up execution by 75%. The adjusted price curve ensures users receive more tokens when base fees decline and corrects execution costs when they rise.

Matcha (0x Labs). Matcha [46] is a DEX aggregator that finds the best executable price by querying over 100 liquidity sources, both public AMMs and private professional market makers, across a growing number of blockchains and building routes that minimize total cost (price impact *plus* gas). Matcha combines AMM liquidity with professional market-maker RFQ quotes, normalizes them to effective prices, and can split a single order across venues to improve depth and slippage. Below, we illustrate a flow that shows how a retail order moves from quote request to on-chain settlement: quotes are sourced in parallel, aggregated, and the best effective price is returned for the trader to sign. This design improves price discovery, reduces slippage, and maintains reliability even

in volatile markets. When a trader requests a quote, the Swap API simultaneously pulls prices from AMM pools and sends RFQ requests to professional market makers. Makers may reply with signed quotes. The API aggregates the responses, compares effective prices (including fees/gas), and returns the best executable quote. If the trader accepts, they sign the order, and the Swap API submits it on-chain for settlement via the 0x Protocol.

KyberSwap Aggregator. KyberSwap [60] is a multi-chain DEX aggregator for trading that unifies fragmented liquidity across AMM and orderbook venues. It comprises on-chain sources with off-chain liquidity, acting as an optimization layer between DEX contracts and order flow, delivering more favorable rates, deeper liquidity, and a smoother trading experience across its multi-chain structure. Its *Dynamic Trade Routing* splits orders and chooses cost-efficient paths in real time to improve effective prices and reduce slippage, while *KyberSwap Limit Orders* are treated as an additional liquidity source - routing through resting orders when advantageous to deepen executable depth. KyberSwap integrates *Professional Market Makers* (PMMs), which collect quotes off-chain and settle on-chain only when they outperform on-chain routes - improving price discovery without introducing extra volatility. Trades can be split and routed across multiple venues within the same network. The aggregator can generate additional gas savings by optimizing routes within KyberSwap pools, minimizing the number of transfers, and reducing the gas fees associated with token trading. Recently, KyberSwap integrated **NEAR Intents** [61], a multi-chain transaction protocol where users specify desired outcomes and let third parties compete to provide the best solution. It is an intent framework for executing asset exchanges and actions across multiple chains and off-chain domains, unlocking deeper cross-chain liquidity, offering better pricing, achieving faster finality, and providing a smoother user experience. NEAR Intents allow trading of any asset on any chain from a single account or by any agent, without the need for bridging or wrapping assets.

ODOS. Odos [62] is a DEX optimization aggregator using *Smart Order Routing* (SOR)[5] to build optimal paths across liquidity sources on 14 different chains. It supports simple swaps, limit orders, and multi-token atomic swaps (up to six inputs into one or multiple outputs at specified proportions). Quotes include gas and expected slippage, guaranteeing the displayed rate matches the executed rate The router optimizes the total execution cost, which combines the price impact and gas. Odos also offers cross-chain routing without manual bridging and a developer API (free tier, higher-volume plans on request). Odos includes *Liquidity Zap*, a liquidity management unit that supports Uniswap V2 style pools, simplifying the process of moving from one liquidity position to another in a single, atomic transaction. Odos calculates the optimal swap amounts to match current pool reserves, converts into the most advantageous LP tokens, and refunds any surplus back to your wallet so only the necessary contribution is used. Using this integration, users can adjust investments in a single step across different liquidity pools or specific tokens. Together with Across Protocol [63], Odos enables intent-based cross-chain swaps that merge multiple source-chain assets into a single destination asset in one transaction: Odos computes the optimal routes, Across handles intent submission while solvers (relayers) compete to fill, and settlement verifies and releases funds on the destination.

CoW Protocol. CoW Protocol [27] is an intent-based DEX aggregator that batches users' signed trade intents and runs a *Fair Combinatorial Batch Auction* (FCBA) where solvers compete to deliver the best overall execution price. FCBA combines batch auction.[6] and multiple simultaneous auctions: solvers submit individual-trade bids and batched bids, but batched bids are considered only if they are better for all traders relative to the outcome of the simultaneous auctions constructed using the

[5]Smart Order Routing is an automated process that optimally trades orders across different trading platforms at the best possible prices and with the highest liquidity. It uses algorithms to evaluate liquidity, fees, prices, and latency. The benefits of SOR include automatically searching for the best possible price, reducing slippage, and increasing trading efficiency.

[6]A batch auction is a mechanism that orders over a time window and executes them simultaneously, effectively combining many orders into a single transaction

individual-trade bids[7] Before using AMMs or external liquidity, solvers attempt to match orders internally through a *Coincidence of Wants* (CoW), a direct, peer-to-peer mechanism where two users, each holding the asset the other needs, exchange assets directly in an equivalent barter, reducing fees and MEV exposure. When no CoW swaps are available, solvers source liquidity on- and off-chain and clear the batch at a single clearing price. The auction-based design shifts price discovery off the mempool into a solver competition, aiming for better prices, fewer failed swaps, and stronger MEV protection compared with single-order routing.

Velora Delta. Velora Delta [65] is an intent-based trading protocol built on the decentralized intents–solving network (Portikus), live on different chain networks such as Ethereum, Base, and Optimism. Delta enables gasless submission where traders sign an order without holding a gas token; agents execute on their behalf and recover costs in the fill price. Delta enables orders to be submitted without a gas token: traders sign off-chain, agents execute on-chain, on their behalf, and fees are recovered in the fill price. Routing orders through the protocol mitigates common MEV attacks, while agent competition pushes toward the best effective rate. An order execution flows through three stages: first, a validated order is forwarded to Portikus through agent coordination. Once the order is received, Portikus runs an auction to determine the winning Agent who will execute the stated intent. Finally, the Agent calls Delta's contracts to complete the swap, including cross-chain paths to settle the requested intent on-chain.

Velora Delta and Velora Market [66] are distinct products with different pipelines, contracts, and APIs. While Velora Delta emphasizes the management of intent with agent auction and optional gasless, cross-chain execution, Velora Market is a classic DEX aggregator that finds paths across AMMs and RFQ makers for traditional swaps and executes a single on-chain router transaction that the user pays gas for.

[7] For further information on this topic, see [64].

UniswapX. UniswapX [36] is a non-custodial, auction-based trading protocol for Ethereum that delegates routing and batching to *fillers* who compete via Dutch auctions and aggregate on-chain and off-chain liquidity to deliver best-available execution. It internalizes MEV in the form of price improvement, offers gas-free swaps, and can be extended to support cross-chain trading. Orders are structured as off-chain signatures where a swapper first approves *Permit2*[8] and then signs an order specifying input/output tokens, input/output amounts, a start price amount, a minimum acceptable amount, a time-based decay function, a claim deadline, and authorization for the on-chain *Reactor* contract, which validates execution against user parameters and reverts when unmet, to spend on the user's behalf. Signed intents are broadcast to an off-chain order book where fillers discover them (via the UniswapX Orders API[9]), simulate routes across AMMs, RFQ makers, and off-chain inventory, and settle the winning path on-chain. The Reactor enforces order constraints and reverts any violations. Fillers pay gas upfront and obtain it back in the execution price, giving swappers a gasless experience. UniswapX supports on-chain-settled Dutch auctions and RFQ flows that guarantee short exclusivity to a selected filler. This design internalizes MEV, offers gasless swaps where possible, and extends to cross-chain settlement with additional components. The same pattern extends to cross-chain swaps: orders remain off-chain signatures, fillers compete to route across domains, and the Reactor enforces user constraints on the settlement chain. Fillers can act as *direct Fillers* that authorize output tokens to the Reactor and submit `execute`/`executeBatch` from an EOA or deploy custom *Executor* contracts that call `executeWithCallback`/ `executeBatchWithCallback` to acquire and approve the output tokens before settlement. In case of order transaction through Priority Gas Auctions (PGA), the protocol uses the `PriorityOrderReactor`, a reactor designed for PGA. Fillers bid during fulfillment by setting custom priority fees. Each order specifies a minimum executable price with no

[8]Permit2 is a token-approval contract introduced by Uniswap that manages ERC-20 permissions via off-chain, gasless signatures. It standardizes approvals across apps and reduces repeated on-chain *approve* calls.

[9]https://api.uniswap.org/v2/uniswapx/docs

start-price auction and a start block after which it becomes fillable. Only the transaction with the highest priority fee is successful, while the others are reverted.

HashFlow. Hashflow [51] is a DEX Aggregator that performs cross-chain swaps. This protocol combines constant-product AMMs with *Request-for-Quote* (RFQ) model where professional market makers quote executable prices. Hashflow utilizes a SOR architecture that offers traders the best prices. Hashflow also supports *bridgeless* cross-chain swaps where users trade assets directly across major chains without wrapped tokens or conventional bridges, reducing trust assumptions. The result is a security-first, user-centric exchange that brings tighter pricing, robust MEV resistance, and seamless multi-chain execution to DeFi.

4.2.5 Other Relevant Infrastructures

Anoma. Anoma [44] is an intent-centric, decentralised counterparty discovery system[10] where users submit intents as unbalanced transactions, and solvers complete them by finding executable paths. These intents enter an intent mempool monitored by solvers, who compose compatible intents (and, if needed, their own liquidity) into balanced transactions. Each balanced transaction must satisfy the attached validity predicates - declarative checks that evaluate and enforce prices, policies, and other constraints. Solver selection can be first-completer or auction-based (with multi-domain scoring for surplus, fees, and latency). Balanced transactions are forwarded to the transaction mempool, from which block producers order them. Executor nodes then verify predicates and apply state transitions deterministically. Applications expose transaction functions that produce either intents or fully specified transactions, all conforming to a standardized transaction object for cross-application composition.

[10] We include this protocol because in 2020, it introduced the concept of intents.

Across. Across [67] is an intents-based, cross-chain protocol[11] where users submit an intent that relayers competitively fill near-instantly on the destination chain; the user's funds are escrowed on the origin chain, and relayers are later repaid after optimistic verification via UMA's oracle[12]. The architecture combines an RFQ layer for pricing and claiming, a fill execution step by solvers, and a settlement layer that batches proofs and repayments - enabling gasless orders and cross-chain swaps while prioritizing capital efficiency, a competitive solver market, and a no-slippage fee model. Relayers must cover gas expenses, the cost of locking capital, and the risk that operational errors or adverse conditions prevent repayment. Users set fees for relayers to incentivize them to execute a bridge transfer. The incentive is the spread between the order's input and output amounts. At fill time, relayers select a repayment chain and are reimbursed by users.

4.2.6 Comparative Analysis and Reference Model

From the protocol-by-protocol analysis, we can detect distinctive phases that characterize the intent-based architecture: *(i)* intent submission, *(ii)* broadcasting and aggregation of intents, *(iii)* auction competition among trusted third-parties to select the best trade, and *(iv)* effective execution and settlement of the intents. In most designs, there is a clear separation between off-chain processes (storage, aggregation, and competition) and on-chain processes (verification, execution, and settlement), which enables richer matching at lower gas costs and minimizes trust preservation through on-chain verification. Starting from this clear composition, we exclude aggregators and bridges that do not have a proposal-selection phase and an action/execution module, and that function purely as routers for single-chain or cross-chain swaps. This keeps the analysis focused on how intent-based engines are built, specifically the units, roles, and interfaces that convert user intents into verifiable execution. Accord-

[11] DeFiLlama categorises it as a cross-chain bridge, so we excluded it from the previous section.

[12] UMA is an optimistic oracle and dispute arbitration system that securely allows for arbitrary types of data to be brought on-chain.

ing to this, we restrict our comparison to six systems (1inch Fusion+, UniswapX, CoW Protocol, Across Protocol, NEAR Intents, and Velora Delta) and contrast their implementations to identify shared patterns and divergences.

First, we normalise terminology: we map the core agents and elements of intent-based architectures and document how each protocol names and treats these components. Table 5 aims at clarifying the glossary and shows that several implementations are functionally overlapping, often differing only in nomenclature or minor details. From the

Table 5: Intent-based protocol components.

Protocol	User surface	Executors	Selection mechanism
1inch Fusion+	Makers	Resolvers	Dutch auction
UniswapX	Swappers	Fillers	Dutch auction
CoW Protocol	Users	Solvers	FCBA
Across Protocol	Users	Relayers	Off-chain RFQ / competition
NEAR Intents	Users / Agents	Solvers	Off-chain RFQ / competition
Velora Delta	Users	Portikus Agents	Off-chain auction

table, the labels for **Executors** vary by implementation, but the role is the same: all of them compete in an off-chain auction to win and execute user intents. Even if the mechanics differ from one protocol to another (i.e. Dutch auction, FCBA, Off-chain RF competition), this competitive layer of the auction unit is fundamental in the intent-based architecture because it enables better trading outcomes, facilitating liquidity concentration and improving price discovery.

Table 6 aims at providing the key factor information about intent management. It traces the full lifecycle and execution path: the **Creation** column lists how intents are expressed and submitted; **Partially fillable** column shows whether the protocol natively supports partial fills or fill-or-kill orders; the **Validity window** column identifies the intent time to live parameter; the **Storage** column indicates where intents are saved before the settlement; the **Multiple-intent Execution** column describes whether distinct intents can be cleared together or only per order;

and **On-chain Settlement** column list how settlement is verified and finalised. The comparison highlights design choices and their trade-offs.

Table 6: Intent management across protocols grouped by phase.

	Phase I — Intent formation			Phase II — Distribution	Phase III — Execution	Phase IV — Settlement
Protocol	**Creation**	**Partially fillable**	**Validity window**	**Storage**	**Multiple-intent Exec.**	**On-chain Settl.**
1inch Fusion+	Off-chain EIP-712 order	✓	✓	Off-chain Orderbook	✗	✓
UniswapX	Off-chain order	✗	✓	Off-chain endpoint	✗	✓
CoW Protocol	Off-chain EIP-712 order	✓	✓	Off-chain Orderbook	✓	✓
Across Protocol	Off-chain RFQ intent	✗	✓	Off-chain RFQ SpokePool	✗	✓
NEAR Intents	Off-chain intent	✗	✓	Off-chain Marketplace	✗	✓
Velora Delta	Off-chain EIP-712 order	✗	✓	Off-chain agent auction	✗	✓

Regarding the first stage, all protocols represent intents as *off-chain orders*. In general, an intent is an off-chain, signed instruction authorizing a trusted third-party to execute a trade on the user's behalf, specifying the input and output assets, a validity window (defined by each protocol's intent schema), and the user-defined minimum output amount and other execution constraints. Most protocols use the EIP-712 [68] Ethereum standard for hashing and signing typed, structured data with a human-readable schema and deterministic encoding. The standard generates digital signatures of structured messages, where the format and content of the message itself are defined through a type schema. It enhances interoperability by unifying message formats across applications and improves security and user experience by making the signed data explicit, thereby reducing misinterpretation and phishing risks. The signer of the message controls what they are signing, avoiding phishing or spoofing attacks. Through digital signature, senders can ensure that the messages they send are not modified or falsified during transmission, and that only authorized recipients can read and understand the content of the messages.

We analyzed the diffusion of the *partial fillability* option across protocols because we consider it a critical factor for execution quality, liquidity utilisation, and overall user experience. Partial fills allow an order to execute incrementally as liquidity becomes available, rather than reverting or waiting for full size. This reduces slippage and price impact, improves fill probability in markets, lowers the risk of revert-induced gas waste, and often yields better realised pricing over time. Only 1inch

Fusion+ and Cow Protocol provide native, protocol-level partial fills[13]. 1inch Fusion+ achieves partial execution via tranching, and CoW Protocol supports partially fillable orders within its batch auctions (increasing the chance of CoW matches, decreasing gas, and avoiding LP fees when matched peer-to-peer). By contrast, most other protocols execute a single intent atomically; partial execution must be emulated client-side (e.g., by splitting an order).

Another relevant aspect that associates all the protocols is the intents aggregation that all of them pursue off-chain including all the intent in an *order book* (which each protocol refer with a specific name) where the intent are store for a spefic amount of time associated to the intents validity time. In this structure, all the solvers are able to visualize the active intents that has to be performed.

Another common aspect is *off-chain aggregation*: intents are pooled in an order book (each protocol names it differently) and retained until their validity window expires. Protocols set explicit *timeouts* for intent validity: the Across Protocol introduces a `fillDeadline` parameter that sets the latest time a fill can occur; the Cow Protocol `validTo` parameter serves the same purpose; and Velora Delta, UniswapX, and others enforce per-order expiry, after which, the intent becomes unfillable and must be resubmitted. By subscribing to the order book, solvers discover and monitor active intents, then compete to fill eligible ones before they expire. The way the solvers compete is one of the aspects that differ most among protocols.

In assessing liquidity improvements, we focus on multiple-intent execution, which aggregates fragmented order flow and clears offsetting trades internally, converting fragmentation into direct crossing liquidity. This decreases slippage and LP fees, eliminates gas across fills, reduces MEV exposure, and enables larger orders that might fail on their own. A uniform clearing price further reduces latency races and adverse selection, encouraging deeper participation and more competitive solver bids. Across the set, only CoW Protocol supports multiple-intent execu-

[13] Across allowed relayers to fill deposits in parts until v2; this option was removed in v3.

tion[14], batching orders for uniform clearing, whereas all the other protocols execute per order without cross-intent aggregation.

Finally, we examine how executors are incentivised to *compete* and *settle* intents. Users define their intent and lock a reward on the source chain. Executors retrieve these intents, find optimal solutions on the destination chain, and execute them. Locking in a reward will make sure that they complete your transaction, as they do not want to lose money on it. From Table 6 it is clear that incentives come from three sources: *(i)* surplus or spread capture that represents the difference between a user's minimum acceptable price and the actual execution; *(ii)* explicit per-quote or per-execution fees embedded in the all-in quote, and *(iii)* executors' fees with repayment after settlement for models that require up-front capital. Many architectures shift gas to executors, who recover the cost in the execution price, so the user experience is effectively gasless. Despite differing mechanics, the common thread is competition

Table 7: Executor rewards mechanisms across intent-based protocols.

Protocol	**Executor rewards**
1inch Fusion+	Resolver captures auction surplus
UniswapX	Filler captures Dutch auction surplus
CoW Protocol	Solver captures surplus + protocol fee on settlement
Across Protocol	Resolver captures auction surplus + protocol fee
NEAR Intents	Solver spread / per-quote fee
Velora Delta	Agent captures surplus + protocol fee

for surplus or fees, paired with gas sponsorship to keep user costs predictable; our analysis focuses in particular on the reward mechanisms for each protocol. In particular, 1inch Fusion+ pays resolvers via auction surplus and utilizes the tranching[15] to reduce price impact. This is a safety-deposit mechanism that incentivizes resolvers to resolve with-

[14] Across Protocol fills per intent; batching is used for solver repayment.

[15] Tranching is the splitting of a single order into smaller slices that resolvers can fill separately as the Dutch auction price decays.

drawals and cover gas. Also in UniswapX, fillers capture the Dutch auction surplus, which is the price that decays toward the user's worst-acceptable rate, sponsoring gas and recouping it in the fill. The CoW Protocol shares batch-clearing surplus with solvers and applies a settlement fee. Batching improves gas efficiency and can avoid LP fees when orders cross. Unlike previous approaches, Across v3 compensates relayers through a fixed relayer fee, which is the spread between the input and output, and is repaid after verification. Relayers price in gas, opportunity cost, and capital at risk, and can choose the repayment chain. NEAR Intents incentivises solvers via per-quote margins in its marketplace, including fees in the accepted orders. Finally, Velora Delta supports either a fee model (e.g., up to 2% partner fee) or a surplus-share model (e.g., 50% of order surplus), with agents sponsoring gas and embedding costs in their all-in price.

Once defined, the main core aspect of the protocols, we frame a reference flow for intent-based trading and map the most prominent protocols to it. Our goal is not to formalize each auction design, but to capture the end-to-end mechanics users experience: *how intents are authored, disseminated, competitively executed (if applicable), and settled*. To accommodate real-world diversity (e.g., cross-chain fills, RFQ vs. batch auctions, SOR-based execution), we slightly generalize the model-wide spread proposed.

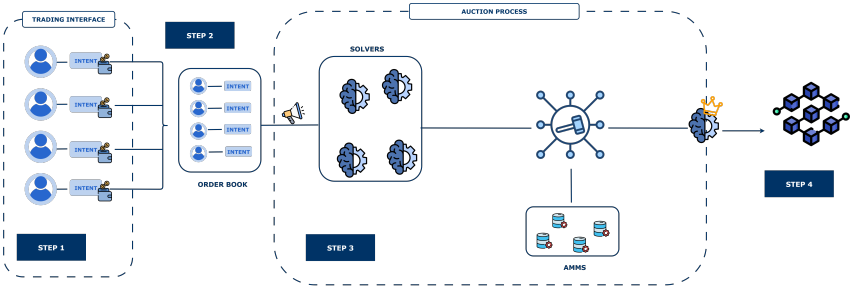

Figure 17: High-level workflow of an intent-based protocol. Users sign and submit intents off-chain; solvers compete to execute them; settlement finalizes on-chain.

Figure 17 describes the reference pipeline we use to *model* intent-based trading systems:

1. **STEP 1: Intent Submission.** The user connects a wallet and either signs an off-chain intent. The message encodes the desired outcome and constraints such as the assets in and out, the price or the slippage bounds, the validity period of the intent, authorizations, and optional privacy flags.

2. **STEP 2: Dissemination & Aggregation.** The protocol makes intents discoverable to solvers through an off-chain orderbook. The goal is consistent discoverability under the protocol's trust and fairness assumptions.

3. **STEP 3: Competition & Selection.** Solvers compete through an auction with a time-based exclusivity window using a deterministic routing algorithm that selects the best route across venues or makers based on user constraints and total cost (price impact plus gas).

4. **STEP 4: Execution & Settlement.** The selected plan is executed atomically on-chain.

The aim of this work is to characterize the end-to-end intent mechanism rather than formally model auction dynamics. Accordingly, we treat auctions and solver behaviour as external and assume rational agents throughout. To maintain transparency, we abstract from participation and bidding frictions; a comprehensive treatment of these costs is deferred to future work.

4.3 A Formal Model of Intent-Based DeFi Protocols

In this section, we present an LTS-based operational model for intent-based protocols, which captures and generalizes the key characteristics informally introduced in the previous section.

After some basic definitions and the description of the configuration shape, we define the inference rules that govern the LTS transitions. Our formalization captures the typical workflow of these protocols (see Figure 17), focusing on the interactions between users, solvers with the protocol. Below, we assume that users and solvers have registered and connected their wallets to the protocol. Wallet connection is required to allow the protocol to settle trades on-chain for users and pay rewards to solvers. Moreover, we assume that solvers act honestly in the sense that their proposed execution plans correctly implement user requests and are compatible with their available balances and prevailing market conditions.

4.3.1 Basic definitions

We assume a set $\mathbb{T}$ of *atomic token types* (ranged over by τ, τ') representing native cryptocurrencies and application-specific tokens. In general, we denote with r, r' nonnegative real numbers ($\mathbb{R}_{\geq 0}$), and we write $r : \tau$ to denote r atomic tokens of type τ ($\tau \in \mathbb{T}$). In addition, we assume the existence of a distinguished token type $\tau_p \in \mathbb{T}$, called the *protocol token*,

which is used by the protocol to reward the winning solver. Rewards paid in τ_p serve as an incentive to encourage solvers to participate in auctions and provide execution plans.

Moreover, we define a set of *agents* $\mathbb{A} = \mathcal{U} \uplus \mathcal{S}$ as the disjoint union of the set of users $\mathcal{U}$ and the set of solvers $\mathcal{S}$, ranged over by U, U' and S, S', respectively. We range over a generic agent with A, A'.

We associate to each agent A a *wallet* denoted by the term $\mathsf{A}[\sigma_\mathsf{A}]$, where $\sigma_\mathsf{A} \in \mathbb{T} \mapsto \mathbb{R}_{\geq 0}$ is a total map representing A's token supplies, namely $\sigma_\mathsf{A}(\tau)$ is the amount of token τ owned by A. We denote with $dom(\sigma_\mathsf{A})$ the set of token types τ where $\sigma_\mathsf{A}(\tau) > 0$. Given a token supply σ, a positive real number v, a token type τ, and an operator $\circ \in \{+, -\}$, we define the operation $\sigma \circ v : \tau$ to increase or decrease the amount of token type τ as follows:

$$\sigma \circ v : \tau = \sigma\{\tau \mapsto \sigma(\tau) \circ v\} \text{ if } \quad \tau \in \ dom(\sigma) \text{ and } \sigma(\tau) \circ v \in \mathbb{R}_{\geq 0}$$

where the notation $\sigma\{\tau \mapsto v\}$ means that we update the entry of the map σ for the token type τ to the value v leaving the other entries unchanged.

We model AMMs as an unordered pair $r_0 : \tau_0, r_1 : \tau_1$, where $r_0 : \tau_0$ and $r_1 : \tau_1$ represent the reserves of tokens $\tau_0, \tau_1 \in \mathbb{T}$, with $\tau_0 \neq \tau_1$. Then, we introduce the notion of on *financial state protocol* Γ, Γ' as finite non-empty compositions of agent wallets and AMMs of different chains. For a chain C, its local state is the parallel composition of account supplies and AMMs:

$$\Gamma \ = \ \mathsf{A}_0^\mathsf{C}[\sigma_{\mathsf{A}_0}^\mathsf{C}] \mid \cdots \mid \mathsf{A}_n^\mathsf{C}[\sigma_{\mathsf{A}_n}^\mathsf{C}] \mid \{r_0^\mathsf{C} : \tau_0^\mathsf{C},\ r_1^\mathsf{C} : \tau_1^\mathsf{C}\} \mid \cdots \mid \{r_w^\mathsf{C} : \tau_w^\mathsf{C},\ r_k^\mathsf{C} : \tau_k^\mathsf{C}\} \tag{4.1}$$

Here, $\mathsf{A}_i^\mathsf{C}[\sigma_{\mathsf{A}_i}^\mathsf{C}]$ denotes account A_i on chain C with balance map $\sigma_{\mathsf{A}_i}^\mathsf{C}$, and each reserve $\{r_a^\mathsf{C} : \tau_a^\mathsf{C},\ r_b^\mathsf{C} : \tau_b^\mathsf{C}\}$ denotes an AMM on C.

We assume that the operator $\mid$ is commutative and associative, and we use the notation $X \mid \Gamma$ when we want to highlight some components X of the chain state. Moreover, for simplicity, we assume the following conditions hold: (i) each agent has a single wallet on a specific chain X, namely, $\mathsf{A}_i^X[\sigma_{\mathsf{A}_i^\mathsf{X}}] \neq \mathsf{A}_j^\mathsf{X}[\sigma_{\mathsf{A}_j^\mathsf{X}}]$ for all $i \neq j$; (ii) distinct AMMs cannot hold exactly the same token type on a chain X, namely,$\{\_ : \tau_i^\mathsf{X}, \_ : \tau_{i'}^\mathsf{X}\} \neq \{\_ : \tau_j^\mathsf{X}, \_ : \tau_{j'}^\mathsf{X}\}$

for all $i \neq j$ and $i' \neq j'$. We assume that Γ remains static with respect to the set of AMMs, meaning that no AMM is created. If we represent a scenario confined to a single chain, we do not attach any superscript to the components of the state configuration. Superscripts are only used when we need to distinguish between different chains. In the scenario described above, all operations occur on the same chain, so no chain superscripts are required in the state representation. An agent A can interact with AMMs by performing a *swap* action of the form $\mathsf{A}\colon \mathsf{swap}(v_0 : \tau_0, v_1 : \tau_1)$. This means that agent A sends v_0 units of token τ_0 to an AMM with reserves $\{r_0 : \tau_0, r_1 : \tau_1\}$, assuming she holds at least v_0 units of τ_0, and asks to receive v_1 units of token τ_1 on the same chain in return. For instance, suppose agent A wants to perform the action $\mathsf{A}\colon \mathsf{swap}(100 : \tau_0, 5 : \tau_1)$ over the AMM $\{1000 : \tau_0, 500 : \tau_1\}$. If A has at least 100 units of τ_0 in her wallet, she can send them to the AMM and receive 5 units of τ_1.

An *intent* request submitted by a user U is a tuple

$$I = \mathsf{U}\colon (a, \tau_x, b, \tau_y, t_{\mathsf{exp}}) \tag{4.2}$$

where $a \in \mathbb{R}_{>0}$ is the amount of token τ_x that U is willing to exchange in order to obtain $b \in \mathbb{R}_{>0}$ amounts of token τ_y and t_{exp} is the intent expiration time. An intent is *live* at time t iff $t < t_{\mathsf{exp}}$.

4.3.2 LSM Semantics

We formalize the workflow of Figure 17 as a labeled transition system (LTS), whose states correspond to protocol configurations, and whose transitions capture the evolution of the system driven by agents' actions.

The protocol proceeds in three stages:

1. The system continuously collects valid, well-formed user intents into an unbounded order book. Invalid or expired submissions are ignored (See **STEP 1: Intent Submission** and **STEP 2: Dissemination & Aggregation** of the Figure 17).

2. Once the round begins, the order book is frozen into a snapshot that every solver sees, and a fixed round deadline is set. From the

start of the round until this deadline, each solver may submit at most one timestamped plan for each intent in the snapshot, saying how it would execute that intent. Any intents that arrive after the round has started are queued for the next round (See **STEP 3: Competition & Selection** of the Figure 17).

3. Once the round deadline expires, the protocol reviews all submitted plans; it selects the best plan for each intent (prioritizing user benefit and, in the case of a tie, the earlier submission) and attempts to execute those transactions on-chain. Each selected plan is checked against the current chain state and balances. If a plan is no longer executable (for instance, due to state changes or insufficient liquidity), it is discarded, even if it was the winning plan. The system state is updated to reflect successful fills; satisfied intents are removed, unplanned or unsuccessfully executed intents roll over to the next round, and the proposal pool is cleared (See **STEP 4: Execution & Settlement** of the Figure 17).

The LTS configurations are triples of the form:

$$\langle \Gamma, \mathcal{I}, \rho \rangle \tag{4.3}$$

The first component Γ is the *current chain state*, as defined in equation (4.1), representing the user wallets and AMM reserves.

The second component $\mathcal{I}$ is an *order book*, an unbounded queue of *intent requests*. Formally, it is inductively defined by the following grammar:

$$\mathcal{I} := \emptyset \mid \mathcal{I} \circ I$$

where $\emptyset$ is the empty order book, and $\mathcal{I} \circ I$ represents the order book obtained by appending intent I to the end of $\mathcal{I}$. We write $|\mathcal{I}|$ to denote the number of intents in $\mathcal{I}$.

The third component ρ is the *round* metadata of the form:

$$\rho = \{\mathsf{Inactive}, \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}), \mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{map\_win})\}$$

where:

- Inactive means no round is in progress (either before start or after being completed),

- Active_Round$(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P})$ denotes an ongoing round with start time t_0, round deadline t_d, the set of solvers $\mathcal{S}$ participating in the round, the frozen intent snapshot $\mathcal{I}$ taken at t_0, and $\mathscr{P}$ is the solver proposals. This is a partial map on pairs $(\mathsf{S}, I) \in \mathcal{S} \times \mathcal{I}$ with entries $\mathscr{P}[\mathsf{S}, I] = \overrightarrow{ep}(\mathsf{S}, I)$ that is the proposal of a solver for intent $I \in \mathcal{I}$ at time t_0. We write $\mathscr{P}[\mathsf{S}, I] = \emptyset$ to mean that solver S has not (yet) submitted a proposal for I. Each execution plan $\overrightarrow{ep} = [a_1, a_2, \ldots, a_k]$ is a finite sequence of swap and/or cross-chain swap actions to be executed on the AMMs (see Section 4.3.3 for a definition of these actions). We denote by $|\mathscr{P}|$ the number of submitted (non-empty) proposals, and $dom(\mathscr{P})$ for the domain of $\mathscr{P}$. While $\rho =$ Active_Round$(\cdot)$, solvers $\mathcal{S}$ may submit at most one proposal per intent $I \in \mathcal{I}$ and only for times $now < t_d$ (where now is the current time).

- Settle$(\Gamma, \mathcal{I}^\star,$ map_win$)$ denotes the start of the settlement phase: the auction round has ended, the chain state Γ keeps track of the state evolution, for each intent in the set $\mathcal{I}^\star$ that received at least one winning proposal, the protocol has fixed a pair of winning maps regarding winning solver and its corresponding plan. We keep track of the chain state Γ, the set of the intents that obtain a winning proposal, and the winning maps. Winning mapping are defined as

$$\text{map\_win} : \mathcal{I}^\star \to \mathcal{S} \times \mathscr{P} \qquad \text{map\_win}(I) = (\mathsf{S}^\star_I, \overrightarrow{ep}^\star_I),$$

  that associates to each intent $I \in \mathcal{I}^\star$ its winning solver and plan. We also record the winner's submission time $t^\star_I \coloneqq t_{\mathsf{S}^\star_I, I}$ to induce an execution order, with earlier submissions preceding later ones. Executing a settlement involves running all winning plans in order, tracking the evolving chain state, and updating the order book. Intents fully satisfied are removed, while intents that could not be satisfied are added to the order book.

For simplicity, we assume that the set of solvers $\mathcal{S}$ is fixed. Thus, the protocol maintains $|\mathcal{S}|$ solver proposals or a subset of this amount (solvers that set their proposal to 0 do not submit any proposal) before selecting the best one and performing the settlement.

The initial configurations of our LTS have the form $\langle \Gamma, \emptyset, \mathsf{Inactive} \rangle$ where Γ is a given chain state and the components representing the order book, the solvers' proposals, and the set of intents not pruned are empty, and the round is set to Inactive.

A transition

$$\langle \Gamma, \mathcal{I}, \rho \rangle \xrightarrow{\alpha} \langle \Gamma, \mathcal{I}', \rho \rangle$$

represents the evolution of the protocols according to Figure 17. We distinguish three types of actions α: user actions, solver actions, and protocol-triggered actions. Specifically, we have the following actions:

- $\mathsf{U}\colon \mathsf{submit}(I)$ indicates that user U submits her intent I to be inserted inside the order book $\mathcal{I}$. This action can trigger three different behaviors depending on the protocol state. If the user has enough tokens of type τ_{sell}, and the tokens she wants to exchange exist in the system, the submission is accepted ([IntentSubmit] rule). Similarly, if the user lacks sufficient tokens of type τ_{sell}, or if the wanted token types are not supported, the submission is rejected ([InvalidIntent] rule). Finally, if an intent arrives while the round is already active (i.e., the system is in state $\mathsf{Active\_Round}(\cdot)$), the intent is still appended to the order book, but it does not belong to the frozen order book used for the current round ([IntentNextRound] rule).

- $\mathsf{startRound}(t_0, t_d, \mathcal{S})$ starts a new round at start time t_0 with the solvers $\mathcal{S}$ until the round deadline t_d. The snapshot of the order book $\mathcal{I}$ is frozen and the proposal map $\mathscr{P}$ is initialized with one empty slot per pair $(\mathsf{S}, I) \in \mathcal{S} \times \mathcal{I}$ ([RoundStart] rule).

- $\mathsf{S} : \mathsf{propose}(\overrightarrow{ep}, I)$ denotes that solver S submits a single execution plan $\overrightarrow{ep}$ for intent I during an active round and before the round deadline t_d ([PlanSubmit] rule). Each solver can make a single submission for a specific intent: each slot (S, I) admits at most one

non-empty plan in a round. Any duplicate, late (after t_d), out-of-snapshot, or ill-formed submission is ignored ([PLANINVALID] rule).

- pick_winners defines for each intent in the frozen order book that has at least one solver proposal, the winning plan - maximizing user benefit and, on ties, preferring earlier submissions - and executing it ([SELECT&WINNERS] rule).

- settle($\overrightarrow{ep}_{\mathsf{S}_I^\star}, I$) executes each winning plan $\overrightarrow{ep}_{\mathsf{S}_I^\star}$ for the intent I, applying the corresponding state transition(s) to produce the updated chain state; satisfied intents are removed and any remaining intents are carried over ([SETTLE] rule). If a winning plan cannot be executed, its entry is removed from win_map and the associated intent is reinserted into the order book so it remains eligible in the next round ([FAIL_SETTLE] rule).

- perform specifies that settlement has fully completed: all winning plans have been executed. The system atomically commits the chain state resulting from those executions, checks the carry-over order book (dropping intents that are no longer admissible), and exits the settlement context. After this action, the system is ready to collect new intents or initiate the next round ([PERFORM] rule).

It is convenient to introduce some auxiliary notations to facilitate the definition of the semantic rules.

Acceptable Intent. We say that the intent I is *acceptable* in a state Γ, in symbols $I \vdash \Gamma$, when the submitting user U has enough selling tokens τ_{sell} in her wallet and there exists an AMM or another user who has tokens of the required assets τ_{buy}, and the intent is still valid, formally:

$$\mathsf{U} : (a : \tau_{sell}, \_ : \tau_{buy}) \vdash \Gamma \iff \sigma_{\mathsf{U}}(\tau_{sell}) \geq a \ \wedge \ \tau_{sell}, \tau_{buy} \in \Gamma \wedge \quad (4.4)$$
$$(\tau_{sell}, \tau_{buy}) \neq \tau_p \wedge \ t_{\mathsf{exp}_I} > now$$

This notion allows us to discard requests that cannot be satisfied because user U does not have enough funds to cover her transaction or because the requested resource is not available, and the intent has expired.

Vice versa, we say that an intent I of user U is *unacceptable* in a state Γ, when $I \vdash \Gamma$ does not hold, written $I \nvdash \Gamma$.

Generic well-formed plan. Suppose we have a fixed a chain state Γ, a frozen snapshot of eligible intents $\mathcal{I}$ taken at time t_0, and a round deadline t_d. We say that a *plan proposal* for solver S a mapping $\mathscr{P}[\mathsf{S}, I] = \overrightarrow{ep}(\mathsf{S}, I)$ is *well-formed* in state Γ relative to snapshot $\mathcal{I}$ and deadline t_d, written

$$\mathsf{WF\_Plan}(\overrightarrow{ep}, I, \mathcal{I})$$

iff all the following conditions hold.

1. Coverage. For every executing $\overrightarrow{ep}$ from Γ achieves (at least) the outcome required by I.

2. Feasibility (resources and liquidity). Each $\overrightarrow{ep} = \langle a_1, \ldots, a_k \rangle$ associated to an intent I is executable in order: *(i)* required balances/allowances exist (accounting for inflows created by earlier a_j); *(ii)* every venue/pool referenced is present in Γ, and reserves remain non-negative, and outputs follow the venue's pricing rule/invariant; *(iii)* any swap/routing step is valid under its venue policy (fees, slippage bounds, path existence).

3. Safety (conservation and non-negativity). All intermediate account balances and pool reserves remain ≥ 0; no double-spend of the same allowance/escrow occurs; no unauthorized transfer is attempted.

4. Consistency across intents. Actions shared across proposal plans are compatible: they do not race on exclusive resources, do not reuse consumed outputs, and collectively induce a single, well-defined post-state.

5. Timeliness. Every $\mathscr{P}[\mathsf{S}, I] = \overrightarrow{ep}$ is intended to complete *before* t_d, and each I is still valid: $now < t_{\mathsf{exp}} \leq t_d$ (as recorded at submission time).

6. Authorization. All signatures, permits, and approvals required by the actions in $\overrightarrow{ep}(\mathsf{S}, I)$ are present and valid.

7. Domain & policy admissibility. The plan $\overrightarrow{ep}$ is *acceptable* in Γ, written $\overrightarrow{ep} \vdash \Gamma$: every referenced asset is supported, and every venue/chain used is permitted by protocol policy and reachable from Γ.

In contrast, we say that a plan proposal $\overrightarrow{ep}$ is *ill-formed* when at least

one clause above fails:

$$\neg\, \mathsf{WF\_Plan}(\overrightarrow{ep}, I, \mathcal{I})$$

Now, we introduce the semantic rules that define the transitions of our LTS. The [IntentSubmit] rule is triggered whenever a user U submits a valid intent I:

$$\text{[IntentSubmit]}\quad \frac{I \vdash \Gamma}{\langle \Gamma, \mathcal{I}, \rho\rangle \xrightarrow{\mathsf{U}:\, \mathsf{submit}(I)} \langle \Gamma, \mathcal{I} \circ I, \rho\rangle}$$

The rule appends the intent I to the order book $\mathcal{I}$, provided that I is acceptable in the current state Γ. Through this rule, the system accumulates these submissions in the order book.

Next, the [InvalidIntent] rule prevents an invalid intent requests from being appended to $\mathcal{I}$ or expired intents, namely, when I fails to satisfy the acceptability conditions in Γ:

$$\text{[InvalidIntent]}\quad \frac{I \nvdash \Gamma}{\langle \Gamma, \mathcal{I}, \rho\rangle \xrightarrow{\mathsf{U}:\, \mathsf{submit}(I)} \langle \Gamma, \mathcal{I}, \rho\rangle}$$

In this case, a state transition occurs, but the configuration of the system remains unchanged as if the action never happened. Note that the two rules above collectively ensure that only valid intents contribute to the auction process.

The system advances to the next phase involving the solvers and triggers the [RoundStart] rule when the round starts, and the parameter ρ is updated to Active_Round at time t_0. At time t_0, the order book $\mathcal{I}$ is frozen as all the solvers have the same list of intent $\mathcal{I}$. In this phase, each solver generates in isolation a possible execution plan using its own strategy for each eligible intent of $\mathcal{I}$ live at t_0 (third premises) until the deadline t_d is reached. The deadline is chosen strictly before the earliest intent expiry to guarantee executability before any intent lapses (fourth premises).

$$\text{[RoundStart]}\quad \frac{\mathcal{I} \neq \emptyset \qquad \mathcal{S} \neq \emptyset \qquad t_0 \;<\; t_d \;<\; \min\{\, t_{\mathsf{exp}} \mid I \in \mathcal{I} \,\}}{\langle \Gamma, \mathcal{I}, \mathsf{Inactive}\rangle \xrightarrow{\mathsf{startRound}(t_0, \mathcal{S}, t_d)} \langle \Gamma, \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}')\rangle}$$

where $\mathscr{P}'[\mathsf{S}, I] = \emptyset$ for all $(\mathsf{S}, I) \in \mathcal{S} \times \mathcal{I}$.

The [PLANSUBMIT] rule governs how a solver submits exactly one execution plan for a given intent while the round is active and the round deadline has not expired. The submission is recorded together with its timestamp t and later bound to the solver's delivery. The premises ensure that the system is in an active round with snapshot $\mathcal{I}$ and deadline t_d, the submitting solver S belongs to the solver set $\mathcal{S}$, the target intent I is part of the frozen snapshot $\mathcal{I}$, the proposal plan $\mathscr{P}[\mathsf{S}, I]$ exists and is still empty; and the proposed plan $\overrightarrow{ep}$ is well-formed for I with respect to $\mathcal{I}$ and the current chain state Γ. Formally:

[PLANSUBMIT]

$$\frac{\begin{array}{c}\rho = \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}) \quad t = now \\ \rho' = \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}[(\mathsf{S}, I) \mapsto \langle \overrightarrow{ep}, t \rangle]) \quad t < t_d \\ \mathsf{S} \in \mathcal{S} \quad I \in \mathcal{I} \qquad \mathscr{P}[\mathsf{S}, I] = \emptyset \quad \mathsf{WF\_Plan}(\overrightarrow{ep}, I, \mathcal{I})\end{array}}{\langle \Gamma, \mathcal{I}', \rho \rangle \xrightarrow{\mathsf{S:propose}(\overrightarrow{ep}, I)} \langle \Gamma, \mathcal{I}', \rho' \rangle}$$

The [PLANINVALID] rule discards proposals that are submitted in the wrong phase (system is Inactive), after the round deadline, by a non-member solver, for an intent outside the snapshot, to a non-existent or already-filled slot, or that fail well-formedness checks. Formally:

[PLANINVALID]

$$\frac{\begin{array}{c}(\rho = \mathsf{Inactive}) \ \vee \ (\rho = \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}) \wedge \\ [\, now \geq t_d \ \vee \ \mathsf{S} \notin \mathcal{S} \ \vee \ I \notin \mathcal{I} \ \vee \ \mathscr{P}[\mathsf{S}, I] \neq \emptyset \ \vee \ \neg \mathsf{wfPlan}(\overrightarrow{ep}, I, \mathcal{I}) \,])\end{array}}{\langle \Gamma, \mathcal{I}', \rho \rangle \xrightarrow{\mathsf{S:propose}(\overrightarrow{ep}, I)} \langle \Gamma, \mathcal{I}', \rho \rangle}$$

When an invalid condition is met, the rule fires without producing any effects: the transition is discarded and the state remains as is.

After the round deadline, the [PICK_WINNERS] rule selects an intent I from the frozen snapshot $\mathcal{I}$.

[PICK_WINNERS]

$$\frac{\begin{array}{c}now \geq t_d \mathcal{I}^\star \ = \ \{\, I \in \mathcal{I} \mid \mathsf{Eligible}(I) \neq \emptyset \,\} \qquad \hat{\mathcal{I}} \ = \ \mathcal{I}' \circ \{\, \mathcal{I} \setminus \mathcal{I}^\star \,\} \\ \forall I \in \mathcal{I}^\star.\ \mathsf{S}_I^\star = \underset{\substack{\mathsf{S} \in \mathsf{Eligible}(I) \\ \mathscr{P}[\mathsf{S}, I] = \langle \overrightarrow{ep}, t \rangle}}{\operatorname{argmax}} \ (F_I(\overrightarrow{ep}, I),\ t_d - t)\end{array}}{\langle \Gamma, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}) \rangle \xrightarrow{\mathsf{pick\_winners}} \langle \Gamma, \hat{\mathcal{I}}, \mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{win\_map}) \rangle}$$

We say that a solver S is *eligible* to win intent I if, during the active round, it submitted a non-empty proposal for I at time t. Formally

$$\mathsf{Eligible}(I) \;=\; \{\, \mathsf{S} \in \mathcal{S} \;\mid\; \mathscr{P}[\mathsf{S}, I] = \langle \overrightarrow{ep}, t \rangle \;\neq\; \emptyset \,\}$$

Winners are chosen only for intents that received at least one proposal ($\mathcal{I}^\star$); intents with no eligible proposals (third premise) are carried over to the next round by appending them to the live book ($\hat{\mathcal{I}}$). Winner selection follows a per-intent maximization: for each I, the protocol chooses the solver-plan pair that maximizes the intent's objective :

$$\mathsf{S}_I^\star \;\in\; \underset{\substack{\mathsf{S} \in \mathsf{Eligible}(I) \\ \mathscr{P}[\mathsf{S},I] = \langle \overrightarrow{ep}, t \rangle}}{\operatorname{argmax}} \; (\, F_I(\overrightarrow{ep}, I),\, t_d - t \,)$$

Equivalently, among the proposals that maximize $F_I(\overrightarrow{ep}, I)$, the protocol breaks ties by earliest submission time (which one can write as maximizing $t_d - t$). The mapping component contains all the winners:

$$\mathsf{win\_map} \;=\; \{\, I \;\mapsto\; (\mathsf{S}_I^\star, \overrightarrow{ep}_{\mathsf{S}_I^\star}) \;\mid\; \mathsf{Eligible}(I) \neq \emptyset \,\},$$

and the system transitions to the settlement context $\mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{win\_map})$, where these winning plans are executed in order while intents without proposals are carried over to the next round.

Next, the [SETTLE] rule is triggered.

[SETTLE]

$$\frac{\Gamma \overset{\overrightarrow{ep}_{\mathsf{S}_I^\star}}{\twoheadrightarrow} \Gamma' \qquad \Gamma \neq \Gamma' \qquad \bar{\mathcal{I}} \;=\; \mathcal{I}^\star \setminus I \qquad \rho' = \mathsf{Settle}(\Gamma', \bar{\mathcal{I}}, \mathsf{win\_map}) \qquad now \geq t_d \qquad I \in \mathcal{I}^\star}{\langle \Gamma, \mathcal{I}', \mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(\overrightarrow{ep}_{\mathsf{S}_I^\star}, I)} \langle \Gamma, \mathcal{I}', \rho' \rangle}$$

It is fired when the round is over (first premise) and there is a winner $I \mapsto \langle \mathsf{S}^\star, \overrightarrow{ep}^\star \rangle$ whose plan, once executed, updates the chain state (second and third premises) in the ρ component. In $\mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{win\_map})$, the Γ carried by ρ acts as a staging state: each winning plan is applied to this working copy and the corresponding entry is removed from $\mathcal{I}^\star$. This design keeps on-chain state changes as atomic as possible while allowing all executable winners to be simulated and composed. The transition

$\Gamma \overset{\overrightarrow{ep}_{S^\star_I}}{\twoheadrightarrow} \Gamma'$ is delegated to an auxiliary LTS (see Section 4.3.3). This LTS interprets the winning plan by executing its action sequence - a concatenation of swap and cross-chain–swap primitives - applying each intermediate update to the state to obtain the final state Γ'.

In the Settle parameter, the proposal is simulated, the state update, and the corresponding intent I is removed from the $\mathcal{I}^\star$ component, while the on-chain configuration state is unchanged.

The [FAIL_SETTLE] rule is triggered when the winning execution plan is non-executable in the auxiliary LTS, and this has no effect ($\Gamma \overset{\overrightarrow{ep}_{S^\star_I}}{\twoheadrightarrow} \Gamma$).

[FAIL_SETTLE]

$$\frac{now \geq t_d \qquad I \in \mathcal{I}^\star \qquad \Gamma \overset{\overrightarrow{ep}_{S^\star_I}}{\twoheadrightarrow} \Gamma \qquad \bar{\mathcal{I}} = \mathcal{I}^\star \setminus I \qquad \rho' = \mathsf{Settle}(\Gamma', \bar{\mathcal{I}}, \mathsf{win\_map})}{\langle \Gamma, \mathcal{I}', \mathsf{Settle}(\Gamma, \mathcal{I}^\star, \mathsf{win\_map})\rangle \xrightarrow{\mathsf{settle}(\overrightarrow{ep}_{S^\star_I}, I)} \langle \Gamma, \mathcal{I}' \circ I, \rho'\rangle}$$

However, the intent I is removed from the set $\mathcal{I}^\star$ and appended to the carry-over order book $\mathcal{I}'$ to be retried in a subsequent active round.

The [PERFORM] rule fires once the settlement phase has concluded, all the winning plans have been executed, and no associated intents remain ($\mathcal{I}^\star = \emptyset$).

[PERFORM]

$$\frac{now \geq t_d \qquad \mathcal{I}'' = \{\, I \in \mathcal{I}' \mid I \vdash \Gamma' \,\}}{\langle \Gamma, \mathcal{I}', \mathsf{Settle}(\Gamma', \emptyset, \emptyset)\rangle \xrightarrow{\mathsf{perform}} \langle \Gamma', \mathcal{I}'', \mathsf{Inactive}\rangle}$$

It atomically commits the staged chain state Γ' (second premise) and exits the settlement context, returning the system to the inactive phase with the carry-over order book $\mathcal{I}'$. The order book is revalidated against the updated state to drop any intents that are no longer admissible (third premise).

Reward mechanism. In intent-based protocols, reward and solver selection functions are critical for aligning incentives, ensuring optimal execution, and maintaining trustworthiness. The reward mechanism of the main intent-based protocols, previously summarized in Table 7, plays a

key role in promoting security, efficiency, and fairness by encouraging solvers to act honestly and effectively, an essential factor for sustaining optimal protocol performance. Although solvers independently employ diverse strategies (e.g., order matching, arbitrage, or MEV minimization), it is the protocol's selection and reward functions that ultimately determine which solver's plan is executed (e.g., *second-price auction* in CoW Protocol).

The selection function assesses proposals based on performance metrics such as cost efficiency, slippage, and user benefit. For example, one can choose the solver's reward in τ_p, as determined by the net surplus generated by the selected solver. In this case, the protocol selects execution plans that maximize the intents contained in the frozen order book $\mathcal{I}$, which records the successful intent exchanges and will be examined in greater detail in the following subsection. This reward is then adjusted to account for the cost of on-chain execution and scaled by a constant factor c as follows:

$$\text{Reward}_{\tau_p} = \text{Surplus}(\mathsf{S}^\star) - c \cdot \text{Fee}_{\text{on-chain}}(\mathsf{S}^\star) + \text{ activationCosts} \quad (4.5)$$

In intent-based protocols, the *reward* and *solver-selection* functions are the core levers for aligning incentives, ensuring good execution, and preserving trust. While solvers may pursue heterogeneous strategies (order matching, arbitrage, MEV minimization, bridging, etc.), it is the protocol's selection and payout rules that ultimately decide whose plan is executed and how value is shared.

4.3.3 Settlement Transition System

We define an auxiliary LTS to capture the concrete token transfers and AMM interactions that the winning solver performs on behalf of the users. Recall that we assume each solver is honest, so its execution plan specifies a well-formed sequence of operations, including token swaps and cross-chain swaps. The states of this transition system are given by the financial states Γ, and any Γ could be an initial state. A transition $\Gamma \overset{\beta}{\twoheadrightarrow} \Gamma'$ represents the execution of a single action in the execution plan of the

solver.

Specifically, the actions that constitute the labels of this LTS are:

- $\mathsf{A}\colon \mathsf{swap}(v_0 : \tau_0, v_1^* : \tau_1)$ denotes submitting v_0 units of τ_0 to the AMM for the pair (τ_0, τ_1) in order to receive *at least* v_1^* units of τ_1 ([SUCC_SWAP] rule). If the preconditions fail and A's τ_0 balance is insufficient or the AMM cannot deliver v_1^* given its reserves and invariant, no trade is executed ([FAIL_SWAP] rule);

- $\mathsf{A}\colon \mathsf{cross\text{-}chainSwap}(v_0 : \tau_0^X, v_1^* : \tau_1^Y)$ denotes initiating a *cross-chain* swap from source chain X to destination chain Y: user A burns v_0 units of τ_0 on X and, after the burn is verified on Y, a solver delivers at least v_1^* units of τ_1 on Y, followed by solver repayment on X ([CC_SWAP] rule). If any precondition fails-e.g., A's balance in τ_0^X is insufficient, the cross-chain proof is invalid or replayed, the AMM on Y cannot produce v_1^* given reserves/invariant, or settlement has already been claimed-no cross-chain trade is executed ([FAIL_CC_SWAP] rule).

The execution of a successful swap transaction is driven by the following rule inspired by the one introduced Bartoletti *et al.* [28] for modeling AMMs:

$$\text{[SUCC\_SWAP]}\quad \frac{\sigma\tau_0 \geq v_0 \quad v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \leq v_1^* \leq v_1}{\mathsf{A}[\sigma]\,|\,\{r_0 : \tau_0, r_1 : \tau_1\}\,|\,\Gamma'' \xrightarrow{\mathsf{A}\colon \mathsf{swap}(v_0:\tau_0,\, v_1^*:\tau_1)} \mathsf{A}[\sigma - v_0 : \tau_0 + v_1 : \tau_1]\,|\,\{r_0 + v_0 : \tau_0, r_1 - v_1 : \tau_1\}\,|\,\Gamma''}$$

The first premise verifies that A's wallet holds at least v_0 units of token τ_0. If this condition is satisfied, both the wallet and the AMM reserves are updated to reflect the token swap. The remaining premises preserve the reserve balance within the AMM, ensuring that the amount v_1 of token τ_1 received, based on the current exchange rate, respects the *constant-product invariant*. Recall that this principle was introduced in Chapter 3 and is captured by the following formula:

$$r_0 \cdot r_1 = (r_0 + v_0) \cdot (r_1 - v_1)$$

The [FAIL_SWAP] rule deals with the case where the liquidity constraints are not met:

[FAIL_SWAP]

$$\frac{\sigma\tau_0 < v_0 \quad v_1 = \dfrac{r_1 \cdot v_0}{r_0 + v_0} \quad 0 \le v_1^* \le v_1}{\mathsf{A}[\sigma]|\,\{r_0 : \tau_0, r_1 : \tau_1\}|\,\Gamma \xrightarrow{\mathsf{A}:\ \mathsf{swap}(v_0:\tau_0, v_1^*:\tau_1)} \mathsf{A}[\sigma]|\,\{r_0 : \tau_0, r_1 : \tau_1\}|\,\Gamma}$$

the first premise ($\sigma\tau_0 < v_0$) checks that the amount of token type τ_0 is less than the exchanged ones, so the action is ignored and the resulting state is left unchanged. The execution of a successful cross-chain swap transaction is driven by the following rule:

[CC_SWAP]

$$\frac{\sigma\tau_0^X \ge v_0^X \;\; \rho\tau_0^Y \ge v_0^Y \quad v_1^Y = \dfrac{r_1 \cdot v_0^Y}{r_0 + v_0^Y} \quad 0 \le v_1^* \le v_1^Y}{\begin{array}{c}\Gamma = \mathsf{A}_X[\sigma_0^X : \tau_0^X]\,|\mathsf{S}_Y[\rho_0^Y : \tau_0^Y]|\,\{r_0^Y : \tau_0^Y,\ r_1^Y : \tau_1^Y\}|\Gamma' \\ \Gamma'' = \mathsf{A}_X[\sigma_0^X - v_0^X : \tau_0^X]|\mathsf{A}_Y[\sigma_1^Y + v_1^Y : \tau_1^Y]|\mathsf{S}_X[\rho_0^X + v_0^X : \tau_0^X]| \\ \mathsf{S}_Y[\rho_0^Y - v_0^Y : \tau_0^Y]|\,\{r_0^Y + v_0^Y : \tau_0^Y, r_1^Y - v_1^Y : \tau_1^Y\}|\,\Gamma'\end{array} \xrightarrow{\mathsf{A}:\ \mathsf{cross\text{-}chainSwap}(v_0^X:\tau_0^X,\ v_1^*:\tau_1^Y)} \Gamma''}$$

The first premise checks that A's wallet on the source chain X holds at least v_0^X units of token τ_0^X to activate the cross-chain procedure; verifies that S's wallet on the destination chain Y has sufficient v_0^Y of token τ_0^Y to perform the market trade. The last premises preserve the reserve balance within the AMM, ensuring that the amount v_1^Y of token τ_1^Y received, based on the current exchange rate, respects the *constant-product invariant*. Once this rule is triggered, A's balance on the source chain X decreases by v_0^X; the solver S exchanges v_0^Y into the AMM on Y and delivers at least v_1^* of τ_1^Y back to the user; AMM reserves update to $(r_0^Y + v_0^Y,\ r_1^Y - v_1^Y)$; and the solver is repaid v_0^X on the source chain X.

The [FAIL_CC_SWAP] rule fires whenever any precondition of the cross-chain swap is violated: if the user's balance of the source token on chain X is insufficient ($\sigma\,\tau_0^X < v_0^X$); if the solver lacks the required liquidity on chain Y ($\rho\,\tau_0^Y < v_0^Y$); if the reserve AMM on the destination chain Y does not preserve the exchange rate; or if the cross-chain proof is invalid or replayed. If any of these conditions hold, the configuration and all balances remain unchanged. This rule preserves safety by preventing

partial fills, dangling repayments, and replayed settlements.

[FAIL_CC_SWAP]

$$\frac{\sigma\tau_0^X < v_0^X \;\; \rho\tau_0^Y < v_0^Y \quad v_1^Y = \dfrac{r_1 \cdot v_0^Y}{r_0 + v_0^Y} \quad 0 \le v_1^* \le v_1^Y}{\mathsf{A}_X[\sigma_0^X : \tau_0^X] \,|\mathsf{S}_Y[\rho_0^Y : \tau_0^Y]|\, \{r_0^Y : \tau_0^Y,\; r_1^Y : \tau_1^Y\}|\Gamma \xrightarrow{\mathsf{A}:\, \mathsf{cross\text{-}chainSwap}(v_0^X \tau_0^X,\, v_1^* \tau_1^Y)} \mathsf{A}_X[\sigma_0^X : \tau_0^X] \,|\mathsf{S}_Y[\rho_0^Y : \tau_0^Y]|\, \{r_0^Y : \tau_0^Y,\; r_1^Y : \tau_1^Y\}|\Gamma}$$

4.4 Application Scenarios

This section illustrates different application scenarios in which we apply the formal model defined in Section 4.3.

4.4.1 First Scenario: Single User Intents

Suppose one user A and three solvers $\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3$ have joined the protocol and connected their wallets. Consider an initial configuration:

$$\langle \Gamma_0, \emptyset, \mathsf{Inactive} \rangle$$

where the chain state Γ_0 contains A's wallet and the AMM reserves of this form:

$$\Gamma_0 = \mathsf{A}[\,20 : \tau_0,\; 20 : \tau_1,\; 20 : \tau_2\,] \mid \{\,100 : \tau_0,\; 200 : \tau_1\,\} \mid \{\,150 : \tau_1,\; 180 : \tau_3\,\} \mid \{\,140 : \tau_3,\; 160 : \tau_2\,\} \mid \{\,100 : \tau_3,\; 250 : \tau_0\,\}.$$

From this configuration, A submits the following acceptable intents to the protocol triggering the [INTENTSUBMIT] rule:

$$I_1 = \mathsf{A}\colon (4, \tau_0, 3, \tau_1, 13),\; I_2 = \mathsf{A}\colon (9, \tau_1, 6, \tau_2, 7),\; \text{and } I_3 = \mathsf{A}\colon (2, \tau_2, 4, \tau_0, 12)$$

$$\langle \Gamma_0, \emptyset, \mathsf{Inactive} \rangle \xrightarrow{\mathsf{A}:\, \mathsf{submit}(I_1)} \langle \Gamma_0, \emptyset \circ I_1, \mathsf{Inactive} \rangle$$

$$\langle \Gamma_0, \emptyset \circ I_1, \mathsf{Inactive} \rangle \xrightarrow{\mathsf{A}:\, \mathsf{submit}(I_2)} \langle \Gamma_0, \emptyset \circ I_1 \circ I_2, \mathsf{Inactive} \rangle$$

$$\langle \Gamma_0, \emptyset \circ I_1 \circ I_2, \mathsf{Inactive} \rangle \xrightarrow{\mathsf{A}:\, \mathsf{submit}(I_3)} \langle \Gamma_0, \emptyset \circ I_1 \circ I_2 \circ I_3, \mathsf{Inactive} \rangle$$

By successive [INTENTSUBMIT] transitions, the order book is extended to $\mathcal{I} = \emptyset \circ I_1 \circ I_2 \circ I_3$. The protocol then opens a round at time $t_0 = 0$. We set the

deadline to $t_d = 6$, which satisfies $t_0 < t_d < \min\{t_{\mathsf{exp}_{I_1}}, t_{\mathsf{exp}_{I_2}}, t_{\mathsf{exp}_{I_3}}\} = 7$, The solver set is $\mathcal{S} = \{\mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3\}$. The configuration evolves as follows:

$$\langle \Gamma_0, \mathcal{I}, \mathsf{Inactive} \rangle \xrightarrow{\mathsf{startRound}(t_0, \mathcal{S}, t_d)} \langle \Gamma_0, \emptyset, \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_0) \rangle,$$

where $\mathscr{P}_0[\mathsf{S}, I] = \emptyset$ for all $(\mathsf{S}, I) \in \mathsf{S} \times \mathcal{I}$. The proposal plan $\mathscr{P}_0$ has the following form:

$$\begin{aligned}
&(\mathsf{S}_1, I_1) = \emptyset,\ (\mathsf{S}_1, I_2) = \emptyset,\ (\mathsf{S}_1, I_3) = \emptyset,\\
&(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \emptyset,\ (\mathsf{S}_2, I_3) = \emptyset,\\
&(\mathsf{S}_3, I_1) = \emptyset,\ (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \emptyset.
\end{aligned}$$

At this step, A submits another intent I_4 of the following form, which triggers the [IntentSubmit] rule:

$$I_4 = \mathsf{A}\colon (3, \tau_1, 4, \tau_3, 15)$$

The intent is accepted as valid; however, the order book contains the intent just arrived ($\mathcal{I}' = I_4$) and will be eligible for the next round:

$$\begin{aligned}
&\langle \Gamma_0, \emptyset, \mathsf{Active_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_0) \rangle \xrightarrow{\mathsf{A}\colon \mathsf{submit}(I_4)}\\
&\langle \Gamma_0, \mathcal{I}', \mathsf{Active_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_0) \rangle
\end{aligned}$$

Each submission satisfies $I \in \mathcal{I}$, $\mathsf{S} \in \mathcal{S}$, $now < t_d$, slot empty, and plan well-formedness. At this point, each solver submits the plan(s) it intends to propose for the current round. Solver S_2 proposes at time t_1 an execution plan, $\overrightarrow{ep}$, for I_2:

$$\begin{aligned}
&\langle \Gamma_0, \mathcal{I}', \mathsf{Active_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_0) \rangle \xrightarrow{\mathsf{S}_2\colon \mathsf{propose}(\overrightarrow{ep}, I_2)}\\
&\langle \Gamma_0, \mathcal{I}', \mathsf{Active_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_1) \rangle
\end{aligned}$$

The proposal plan is updated to $\mathscr{P}_1$ as follows:

$$\begin{aligned}
&(\mathsf{S}_1, I_1) = \emptyset, (\mathsf{S}_1, I_2) = \emptyset,\ (\mathsf{S}_1, I_3) = \emptyset,\\
&(\mathsf{S}_2, I_1) = \emptyset, (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1 \rangle,\ (\mathsf{S}_2, I_3) = \emptyset,\\
&(\mathsf{S}_3, I_1) = \emptyset, (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \emptyset.
\end{aligned}$$

After that, solver S_1 submits at time t_2 the execution plan, $\overrightarrow{ep}$, per intent in I_1, updating the overall configuration as follows:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_1)\rangle \xrightarrow{\mathsf{S}_1:\mathsf{propose}(\overrightarrow{ep}, I_1)} \langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_2)\rangle$$

The proposal plan is updated to $\mathscr{P}_2$ as follows:

$$\begin{aligned}&(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_2\rangle,\ (\mathsf{S}_1, I_2) = \emptyset,\ (\mathsf{S}_1, I_3) = \emptyset,\\ &(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1\rangle,\ (\mathsf{S}_2, I_3) = \emptyset,\\ &(\mathsf{S}_3, I_1) = \emptyset,\ (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \emptyset.\end{aligned}$$

Then, solver S_1 proposes at time t_3 an execution plan, $\overrightarrow{ep}$, for I_2:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_2)\rangle \xrightarrow{\mathsf{S}_1:\mathsf{propose}(\overrightarrow{ep}, I_2)} \langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_3)\rangle$$

The proposal plan is updated to $\mathscr{P}_3$ as follows:

$$\begin{aligned}&(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_2\rangle,\ (\mathsf{S}_1, I_2) = \langle \overrightarrow{ep}, t_3\rangle,\ (\mathsf{S}_1, I_3) = \emptyset\\ &(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1\rangle,\ (\mathsf{S}_2, I_3) = \emptyset\\ &(\mathsf{S}_3, I_1) = \emptyset,\ (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \emptyset\end{aligned}$$

Following this, solver S_3 proposes at time t_4 an execution plan, $\overrightarrow{ep}$, for I_3:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_3)\rangle \xrightarrow{\mathsf{S}_3:\mathsf{propose}(\overrightarrow{ep}, I_3)} \langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_4)\rangle$$

The proposal plan is updated to $\mathscr{P}_4$ as follows:

$$\begin{aligned}&(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_2\rangle,\ (\mathsf{S}_1, I_2) = \langle \overrightarrow{ep}, t_3\rangle,\ (\mathsf{S}_1, I_3) = \emptyset,\\ &(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1\rangle,\ (\mathsf{S}_2, I_3) = \emptyset,\\ &(\mathsf{S}_3, I_1) = \emptyset,\ (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \langle \overrightarrow{ep}, t_4\rangle.\end{aligned}$$

Finally, solver S_1 at time t_5 submits the execution plan $\overrightarrow{ep}$ for I_3.

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_4)\rangle \xrightarrow{\mathsf{S}_1:\mathsf{propose}(\overrightarrow{ep}, I_3)} \langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_5)\rangle$$

The proposal plan is updated to $\mathscr{P}_5$ as follows:

$$(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_2 \rangle,\ (\mathsf{S}_1, I_2) = \langle \overrightarrow{ep}, t_3 \rangle,\ (\mathsf{S}_1, I_3) = \langle \overrightarrow{ep}, t_5 \rangle,$$
$$(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1 \rangle,\ (\mathsf{S}_2, I_3) = \emptyset,$$
$$(\mathsf{S}_3, I_1) = \emptyset,\ (\mathsf{S}_3, I_2) = \emptyset,\ (\mathsf{S}_3, I_3) = \langle \overrightarrow{ep}, t_4 \rangle.$$

When $now \geq t_d$, the system closes the auction window and the component $\mathscr{P}_5$ and the non-empty entries are the following:

$$(\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_1 \rangle, (\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_2 \rangle, (\mathsf{S}_1, I_2) = \langle \overrightarrow{ep}, t_3 \rangle,$$
$$(\mathsf{S}_3, I_3) = \langle \overrightarrow{ep}, t_4 \rangle, (\mathsf{S}_1, I_3) = \langle \overrightarrow{ep}, t_5 \rangle.$$

The component $\mathscr{P}_5$ contains for each intent $I \in \mathcal{I}$, the eligible solvers and their proposals evaluated under the frozen reserves at t_0:

$$\mathsf{Eligible}(I) = \{\, \mathsf{S} \in \mathcal{S} \mid (\mathsf{S}, I) \in dom(\mathscr{P}_5),\ \mathscr{P}_5[\mathsf{S}, I] = \overrightarrow{ep}_{\mathsf{S}} \neq \emptyset \,\}.$$

At this step, we assume to calculate the winners of this round. The winner is $\mathsf{S}_I^\star = \operatorname{argmax}_{\substack{\mathsf{S} \in \mathsf{Eligible}(I) \\ \mathscr{P}[\mathsf{S}, I] = \langle \overrightarrow{ep}, t \rangle}} (\, F_I(\overrightarrow{ep}, I),\ t_d - t\,)$, where F_I ranks proposals by user output.

Because proposals for a given intent have distinct gains, the tie-breaking term based on submission time $t_{\mathsf{S},I}$ is never invoked and can be ignored.

The set of intents that have a winning solution is equal to the frozen set $\mathcal{I}$, and the win_map component, according to the arrival time, is composed of:

$$\mathsf{win\_map} = [I_2 \mapsto (\mathsf{S}_2, \overrightarrow{ep}_{\mathsf{S}_2}), I_1 \mapsto (\mathsf{S}_1, \overrightarrow{ep}_{\mathsf{S}_1}), I_3 \mapsto (\mathsf{S}_3, \overrightarrow{ep}_{\mathsf{S}_3})].$$

The protocol triggers the [PICK&WINNERS] rule:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_5) \rangle \xrightarrow{\mathsf{pick\_winners}}$$
$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_0, \mathcal{I}, \mathsf{win\_map}) \rangle$$

All the intents are then performed in sequence, once the intent set becomes empty. The configuration evolves by performing all the winning

proposals in sequence, decreasing at each settlement step the $\mathcal{I}$ component once the execution of a specific intent is successfully performed. During this step, the $\mathcal{I}$ component is updated to $\hat{\mathcal{I}}' = \mathcal{I} \setminus I_2$.

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_0, \mathcal{I}, \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(I_2, \overrightarrow{ep}_{\mathsf{S}_2})} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_1, \hat{\mathcal{I}}', \mathsf{win\_map}) \rangle$$

The first intent to be processed according to the win_map is I_2 performed by S_2 and consists of the following swap action triggering twice [SWAP] rule: the first time over the reserve $\{150 : \tau_1,\ 180 : \tau_3\}$ and then over the reserve $\{140 : \tau_3,\ 160 : \tau_2\}$:

$$\overrightarrow{ep} = \langle \mathsf{A} \colon \mathsf{swap}(9 : \tau_1,\ 10.19 : \tau_3),\ \mathsf{A} \colon \mathsf{swap}(10.19 : \tau_3,\ 10.85 : \tau_2) \rangle.$$

The first swap action implies an evolution of the reserve $\{150 : \tau_1,\ 180 : \tau_3\}$ of this type $v_1 = \frac{180 \cdot 9}{150+9} = \frac{1620}{159} = \frac{540}{53} = 10.19$ of τ_3 that updates the reserve to:

$$\{159 : \tau_1,\ 180 - \tfrac{540}{53} : \tau_3\} = \{159 : \tau_1,\ 169.81 : \tau_3\}.$$

The second swap implies an evolution of the reserve $\{140 : \tau_3,\ 160 : \tau_2\}$with the intermediate input $v_1 = 10.19 : \tau_3$:

$$v_2 = \frac{160 \cdot 10.19}{140 + 10.19} = \frac{2160}{199} = 10.85\ (\geq 6),$$

Intent I_2 requires at least 6 of τ_2 and the actual receive is $10.85 \geq 6$, hence I_2 is satisfied. The reserve is updated to:

$$\{140 + \tfrac{540}{53} : \tau_3,\ 160 - \tfrac{2160}{199} : \tau_2\} \approx \{150.19 : \tau_3,\ 149.20 : \tau_2\}.$$

The chain state is updated to:

$$\Gamma_1 = \mathsf{A}[\,20 : \tau_0,\ 11 : \tau_1,\ 30.85 : \tau_2\,] \mid \{\,159 : \tau_1,\ 169.81 : \tau_3\,\} \mid \{150.19 : \tau_3,\ 149.20 : \tau_2\,\} \mid \Gamma$$

For intent I_1, the winning solver is S_1. During this step, the $\hat{\mathcal{I}}'$ component is updated to $\hat{\mathcal{I}}' = \hat{\mathcal{I}}' \setminus I_1$.

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_1, \hat{\mathcal{I}}', \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(I_1, \overrightarrow{ep}_{\mathsf{S}_1})} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_2, \hat{\mathcal{I}}'', \mathsf{win\_map}) \rangle$$

The execution plan $\overrightarrow{ep}$ consists of the following swap action triggering the [SWAP] rule that involves the reserve $\{100 : \tau_0,\ 200 : \tau_1\}$:

$$\overrightarrow{ep} = \langle\, \mathsf{A}\colon \mathsf{swap}(4 : \tau_0,\, 3 : \tau_1) \,\rangle.$$

This action allows user A to obtain 7.69 of τ_1, computed as $v_1 = \frac{200 \cdot 4}{100+4} = \frac{800}{104} = 7.69$ in the premise of [SWAP] rule, satisfying I_1 ($7.69 \geq 3$). The reserve is updated to $\{104 : \tau_0,\ 192.31 : \tau_1\}$. Intent I_1 requires at least 3 of τ_1 and the actual receive is $\frac{100}{13} = 7.69 \geq 3$, hence I_1 is satisfied. The chain state is updated to:

$$\Gamma_2 = \mathsf{A}[\, 16 : \tau_0,\ 18.69 : \tau_1,\ 30.85 : \tau_2 \,] \mid \{\, 104 : \tau_0,\ 192.31 : \tau_1 \,\} \mid \Gamma'$$

For intent I_3, the winner solver is S_3. During this step, the set of intents to be processed becomes an empty component ($\hat{\mathcal{I}}'' \setminus I_3 = \emptyset$) .

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_2, \hat{\mathcal{I}}'', \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(I_3, \overrightarrow{ep}_{\mathsf{S}_3})} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_3, \emptyset, \emptyset) \rangle$$

The execution plan $\overrightarrow{ep}$ consists of two swap actions, each triggering the [SWAP] rule.

$$\overrightarrow{ep} \;=\; \langle\, \mathsf{A}\colon \mathsf{swap}(2 : \tau_2,\, 1.729 : \tau_3),\ \mathsf{A}\colon \mathsf{swap}(1.729 : \tau_3,\, 4.25 : \tau_0) \,\rangle.$$

The proposal is evaluated under the chain state Γ_0 and the settlement phase may alter the realized amounts of the swaps. In particular, slippage can modify the reserve amount. However, after executing all intents sequentially, the resulting chain state is slightly different. The received amount of the first swap is higher than quoted - this is allowed by our swap semantics, which only require the received amount to be at

least a lower bound that the user specifies. In this case, the realized outcome exceeds that lower bound, and the proposal is updated as follows:

$$\overrightarrow{ep}_{\mathsf{up}} \;=\; \big\langle\, \mathsf{A}\colon \mathsf{swap}(2:\tau_2,\, 1.98:\tau_3),\; \mathsf{A}\colon \mathsf{swap}(1.729:\tau_3,\, 4.25:\tau_0)\,\big\rangle.$$

Intuitively, the first swap's realized output increased from 1.729 to $1.98\ \tau_3$, and the second swap consumes this updated amount.

In detail, the first swap is done over the reserve $\{150, 19:\tau_3,\ 149, 20:\tau_2\}$ and then over the reserve $\{100:\tau_3,\ 250:\tau_0\}$. The execution of those swaps is composed of the following steps: the first swap achieves the state: $v_2 = \frac{150.17\cdot 2}{149.20+2} = \frac{300.34}{151.20} \approx 1.98 \geq 1.729$ and the second is $v_3 = \frac{250\cdot 1.729}{101.729} = \frac{442.25}{101.729} \approx 4.25$ of token type τ_0).

However, once the last winning plan has been executed, the configuration evolves to:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_2, \hat{\mathcal{I}''}, \mathsf{win\_map})\rangle \xrightarrow{\mathsf{settle}(I_3, \overrightarrow{ep}_{\mathsf{S}_3})} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_3, \emptyset, \emptyset)\rangle$$

The resulting financial state is:

$$\begin{aligned}\Gamma_3 = \Gamma^\star = \mathsf{A}[\,&20.25:\tau_0,\ 18.69:\tau_1,\ 28.85:\tau_2,\ 0.25:\tau_3\,] \mid \mathsf{S}_1\,[\,1:\tau_p\,] \mid \mathsf{S}_2\,[\,1:\tau_p\,] \mid \\ &\mathsf{S}_3\,[\,1:\tau_p\,] \mid \{\,104:\tau_0,\ 192.31:\tau_1\,\} \mid \{100:\tau_2,\ 210:\tau_0\} \mid \\ &\{\,159:\tau_1,\ 169.81:\tau_3\,\} \mid \{151.20:\tau_2,\ 148.19:\tau_3\,\} \mid \\ &\{101.729:\tau_3,\ 245.75:\tau_0\}\end{aligned}$$

Let τ_p denote the protocol's reward unit. We credit integer rewards consistent with one win per solver:

$$\mathsf{S}_1[\,1:\tau_p\,],\ \mathsf{S}_2[\,1:\tau_p\,],\ \mathsf{S}_3[\,1:\tau_p\,]$$

The system settles the winning execution plans and returns the new chain state and the updated order book, $\mathcal{I}'$ contains I_4 because once all the proposals have been performed in sequence.

The final configuration is:

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma^\star, \emptyset, \emptyset)\rangle \xrightarrow{\mathsf{perform}} \langle \Gamma^\star, \mathcal{I}', \mathsf{Inactive}\rangle$$

which completes the batch with a *split-winner* outcome in which each solver wins one intent: S_1 wins I_1, S_2 wins I_2, and S_3 wins I_3.

4.4.2 Second Scenario: Cross-chain Intent

Suppose three users A, B, and C and two solvers S_1, S_2 have joined the protocol and connected their wallets (where τ_i^X on chain X and τ_i^Y on chain Y). The initial configuration is:

$$\langle \Gamma_0,\ \emptyset,\ \mathsf{Inactive} \rangle$$

with chain state:

$$\begin{aligned}\Gamma_0 \ =\ & \mathsf{A}_X[\,10 : \tau_0^X\,] \mid \mathsf{A}_Y[\,5 : \tau_1^Y\,] \mid \mathsf{B}_Y[\,5 : \tau_0^Y,\ 10 : \tau_1^Y\,] \mid \mathsf{C}_Y[\,5 : \tau_1^Y\,] \mid \\ & \mathsf{S}_{1Y}[\,10 : \tau_0^Y\,] \mid \{\,100 : \tau_0^Y,\ 200 : \tau_1^Y\,\} \mid \{\,100 : \tau_1^Y,\ 200 : \tau_2^Y\,\} \mid \\ & \{\,100 : \tau_2^Y,\ 200 : \tau_0^Y\,\}\end{aligned}$$

From this configuration, users submit the following acceptable intents (the first one is cross-chain from X to Y), triggering [IntentAppend] three times:

$$\begin{aligned}& I_1 = \mathsf{A}\colon (4, \tau_0^X, 3, \tau_1^Y, 8), \quad I_2 = \mathsf{B}\colon (3, \tau_1^Y, 4, \tau_0^Y, 10), \\ & I_3 = \mathsf{C}\colon (2, \tau_1^Y, 3, \tau_2^Y, 15)\end{aligned}$$

$$\langle \Gamma_0, \emptyset, \mathsf{Inactive}\rangle \xrightarrow{\mathsf{A}\colon \mathsf{submit}(I_1)} \langle \Gamma_0, \emptyset \circ I_1, \mathsf{Inactive}\rangle \xrightarrow{\mathsf{B}\colon \mathsf{submit}(I_2)}$$

$$\langle \Gamma_0, \emptyset \circ I_1 \circ I_2, \mathsf{Inactive}\rangle \xrightarrow{\mathsf{C}\colon \mathsf{submit}(I_3)} \langle \Gamma_0, \mathcal{I}, \mathsf{Inactive}\rangle$$

so that $\mathcal{I} = \emptyset \circ I_1 \circ I_2 \circ I_3$.

The protocol, triggering [StartRound] rule, opens a round at time t_0 with deadline t_d equal to 7 and solver set $\mathcal{S} = \{\mathsf{S}_1, \mathsf{S}_2\}$; the order book $\mathcal{I}$ is frozen:

$$\begin{aligned}& \langle \Gamma_0,\ \mathcal{I},\ \mathsf{Inactive} \rangle \xrightarrow{\mathsf{startRound}(t_0, \mathcal{S}, t_d)} \\ & \langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_0) \rangle\end{aligned}$$

with $[\mathscr{P}_0(\mathsf{S}, I)] = \emptyset$ for all $(\mathsf{S}, I) \in \mathcal{S} \times \mathcal{I}$:

$$\begin{aligned}& (\mathsf{S}_1, I_1) = \emptyset, \quad (\mathsf{S}_1, I_2) = \emptyset, \quad (\mathsf{S}_1, I_3) = \emptyset, \\ & (\mathsf{S}_2, I_1) = \emptyset, \quad (\mathsf{S}_2, I_2) = \emptyset, \quad (\mathsf{S}_2, I_3) = \emptyset.\end{aligned}$$

At this point, solver S_1 proposes an execution plan $\overrightarrow{ep}$ for the first I_1.

$$\begin{aligned}& \langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I},\ \mathscr{P}_0)\rangle \xrightarrow{\mathsf{S}_1\colon \mathsf{propose}(\overrightarrow{ep}, I_1)} \\ & \langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I},\ \mathscr{P}_1) \rangle\end{aligned}$$

The proposal map is updated to $\mathscr{P}_1$ for the intent I_1:

$$(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_1 \rangle,\ (\mathsf{S}_1, I_2) = \emptyset,\ (\mathsf{S}_1, I_3) = \emptyset,$$
$$(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \emptyset,\ (\mathsf{S}_2, I_3) = \emptyset.$$

At time t_2, solver S_2 proposes an execution plan $\overrightarrow{ep}$ for I_2:

$$\langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_1) \rangle \xrightarrow{\mathsf{S}_2 : \mathsf{propose}(\overrightarrow{ep}, I_2)}$$
$$\langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_2) \rangle$$

The third component of the configuration is updated to $\mathscr{P}_2$ for the intent I_2:

$$(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_1 \rangle,\ (\mathsf{S}_1, I_2) = \emptyset,\ (\mathsf{S}_1, I_3) = \emptyset,$$
$$(\mathsf{S}_2, I_1) = \emptyset,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_2 \rangle,\ (\mathsf{S}_2, I_3) = \emptyset.$$

At t_3, solver S_1 submits the execution plan, $\overrightarrow{ep}$, for intent I_3, updating the overall configuration as follows:

$$\langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_2) \rangle \xrightarrow{\mathsf{S}_1 : \mathsf{propose}(\overrightarrow{ep}, I_3)}$$
$$\langle \Gamma_0,\ \mathcal{I},\ \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_3) \rangle$$

The third component of the configuration is updated to $\mathscr{P}_3$ for the intent I_3:

$$(\mathsf{S}_1, I_1) = \langle \overrightarrow{ep}, t_1 \rangle,\ (\mathsf{S}_2, I_2) = \langle \overrightarrow{ep}, t_2 \rangle,\ (\mathsf{S}_1, I_3) = \langle \overrightarrow{ep}, t_3 \rangle.$$

For each $I \in \mathcal{I}$, define:

$$\mathsf{Eligible}(I) = \{\, \mathsf{S} \in \mathcal{S} \mid (\mathsf{S}, I) \in dom(\mathscr{P}_3),\ \mathscr{P}_3[\mathsf{S}, I] = \overrightarrow{ep}_{\mathsf{S}} \neq \emptyset \,\}.$$

Winners are chosen as $\mathsf{S}_I^\star = \operatorname{argmax}_{\substack{\mathsf{S} \in \mathsf{Eligible}(I) \\ \mathscr{P}[\mathsf{S}, I] = \langle \overrightarrow{ep}, t \rangle}} (\, F_I(\overrightarrow{ep}, I),\ t_d - t\,)$, evaluating outputs under the frozen reserves at t_0. The set of intents that have a winning solution is equal to the frozen set $\mathcal{I}$, and the win_map component, according to the arrival time, is composed of:

$$\mathsf{win\_map} = [I_1 \mapsto (\mathsf{S}_1, \overrightarrow{ep}), I_2 \mapsto (\mathsf{S}_2, \overrightarrow{ep}), I_3 \mapsto (\mathsf{S}_1, \overrightarrow{ep})].$$

The protocol triggers the [PICK&WINNERS] rule, and the system closes the auction window and evolves.

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Active\_Round}(t_0, t_d, \mathcal{S}, \mathcal{I}, \mathscr{P}_3) \rangle \xrightarrow{\mathsf{pick\_winners}} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_0, \mathcal{I}, \mathsf{win\_map}) \rangle$$

All the intents are then performed in sequence, once the intent set becomes empty. The configuration evolves by performing all the winning proposals in sequence, decreasing at each settlement step the $\mathcal{I}$ component once the execution of a specific intent is successfully performed. During this step, the $\mathcal{I}$ component is updated to $\hat{\mathcal{I}}' = \mathcal{I} \setminus I_1$.

$$\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_0, \mathcal{I}, \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(I_1, \overrightarrow{ep}_{\mathsf{S}_1})} \langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_1, \hat{\mathcal{I}}', \mathsf{win\_map}) \rangle$$

The execution plan $\overrightarrow{ep}$ of the intent I_1 submitted by the solver S_1 is composed of a cross-chain swap that triggers the [CROSSCHAINSWAP] rule of the auxiliary transition system. In this scenario, the chain state will be updated to Γ_1 because user A has submitted a cross-chain swap, and to start the cross-chain process, she had to burn her selling amount. Her wallet on chain X has to be updated from $\mathsf{A}_X[\,10 : \tau_0^X\,]$ to $\mathsf{A}_X[\,6 : \tau_0^X\,]$.

$$\overrightarrow{ep} = \langle\ \mathsf{A}\colon \mathsf{cross\text{-}chainSwap}(4\colon\tau_0^X,\ 3\colon\tau_1^Y)\ \rangle$$

User A has already burned her selling cross-chain swap amount. After this step, the cross-chain procedure consists of two main steps: the solver makes a proposal for this intent, performing a swap action on the destination chain using the AMM reserve that contains the token type τ_1^Y defined in I_1 and giving back the outcome to the user. We want to remember that the AMM reserve the solver chooses is strictly related to its performance strategy. The solver could make a direct swap over an AMM that contains the destination token type of the intent or perform several swaps over the different reserves and obtain the final token type at the end. At this stage of writing, we are relaxing the presence of fees. However, if these components are there, the use of multi-hop implies the need to pay more fees for each AMMs.

Solver S_1 chooses the reserve $\{100 : \tau_0^Y,\ 200 : \tau_1^Y\}$ to exchange $4\ \tau_0^Y$ for at least $3\ \tau_1^Y$. Once the exchange rate is calculated, the actual amount of τ_1^Y received is $v_1 = \frac{200 \cdot 4}{100+4} = \frac{800}{104} = \frac{100}{13} \approx 7.6923$ of token type τ_1^Y that is greater than 3 and the updated reserve is $\{104 : \tau_0^Y,\ 200 - 7.69 : \tau_1^Y\}$. Reserve update on Y:

$$\{100 : \tau_0^Y, 200 : \tau_1^Y\} \ \rightarrow\ \left\{104 : \tau_0^Y,\ 192.31 : \tau_1^Y\right\}.$$

The chain state is updated to:

$$\begin{aligned}\Gamma_1 \ =\ & \mathsf{A}_X[\,6 : \tau_0^X\,] \mid \mathsf{A}_Y[\,12.692 : \tau_1^Y\,] \mid \mathsf{B}_Y[\,5 : \tau_0^Y,\ 10 : \tau_1^Y\,] \mid \mathsf{C}_Y[\,5 : \tau_1^Y\,] \mid \\ & \mathsf{S1}_Y[\,6 : \tau_0^Y\,] \mid \{\,104 : \tau_0^Y,\ 192.31 : \tau_1^Y\,\} \mid \Gamma'\end{aligned}$$

For intent I_2, the winning solver is S_2. During this step, the set of intents waiting to be processed is updated to $I_3 = \hat{\mathcal{I}}' \setminus I_2$.

$$\begin{aligned}&\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_1, \hat{\mathcal{I}}', \mathsf{win\_map})\rangle \xrightarrow{\mathsf{settle}(I_2, \overrightarrow{ep}_{\mathsf{S}_2})} \\ &\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_2, I_3, \mathsf{win\_map})\rangle\end{aligned}$$

The execution plan $\overrightarrow{ep}$ triggers twice the [SWAP] rule: first over the reserve $\{100 : \tau_1^Y,\ 200 : \tau_2^Y\}$ and then over the reserve $\{100 : \tau_2^Y,\ 200 : \tau_0^Y\}$:

$$\overrightarrow{ep} \ =\ \langle\, \mathsf{B}\colon \mathsf{swap}(3 : \tau_1^Y,\ 5.8252 : \tau_2^Y),\ \mathsf{B}\colon \mathsf{swap}(5.8252 : \tau_2^Y,\ \tfrac{1200}{109} : \tau_0^Y)\,\rangle.$$

The first action allows user B to obtain $\frac{600}{103} \approx 5.8252$ of τ_2^Y, computed as

$$v_1 \ =\ \frac{200 \cdot 3}{100 + 3} \ =\ \frac{600}{103} \approx 5.8252$$

The reserve of the intermediate hop is updated to

$$\{\,100 + 3 : \tau_1^Y,\ 200 - 5.8252 : \tau_2^Y\,\} \ \approx\ \{\,103 : \tau_1^Y,\ 194.17 : \tau_2^Y\,\}.$$

With the amount obtained by the first swap, the solver exchanges 5.8252 token τ_2^Y into the pool $\{100 : \tau_2^Y,\ 200 : \tau_0^Y\}$ obtaining ≈ 11.0092 of τ_0^Y, computed as

$$v_2 \ =\ \frac{200 \cdot 5.8252}{105.8252} \approx 11.0091\ \tau_0^Y$$

in the premise of [SWAP] rule. The reserve is updated to

$$\{\, 105.83 : \tau_2^Y,\ 189.91 : \tau_0^Y \,\}.$$

Intent I_2 requires at least 4 of τ_0^Y; the actual receive is $\approx 11.0092 \geq 4$, hence I_2 is satisfied. The chain state is updated to:

$$\begin{aligned}\Gamma_2 \;=\; & \mathsf{A}_X[\,6 : \tau_0^X\,] \mid \mathsf{A}_Y[\,12.692 : \tau_1^Y\,] \mid \mathsf{B}_Y[\,16.01 : \tau_0^Y,\ 7 : \tau_1^Y\,] \mid \mathsf{C}_Y[\,5 : \tau_1^Y\,] \mid \\ & \mathsf{S}_{1Y}[\,6 : \tau_0^Y\,] \mid \{\, 104 : \tau_0^Y,\ 192.31 : \tau_1^Y \,\} \mid \{\, 103 : \tau_1^Y,\ 194.17 : \tau_2^Y \,\} \mid \\ & \{\, 105.83 : \tau_2^Y,\ 189.91 : \tau_0^Y \,\} \mid \Gamma''\end{aligned}$$

For intent I_3, the winning solver is S_1, however, the [FAIL_SETTLE] rule is triggered because during settlement, the execution plan becomes infeasible. The configuration evolves to:

$$\begin{aligned}&\langle \Gamma_0, \mathcal{I}', \mathsf{Settle}(\Gamma_2, I_3, \mathsf{win\_map}) \rangle \xrightarrow{\mathsf{settle}(I_3, \overrightarrow{ep}_{\mathsf{S}_1})} \\ &\langle \Gamma_0, \mathcal{I}'', \mathsf{Settle}(\Gamma_2, \emptyset, \emptyset) \rangle\end{aligned}$$

The execution plan consists of the following swap action triggering the [SWAP] rule that involves the reserve $\{\, 103 : \tau_1^Y,\ 194.17 : \tau_2^Y \,\}$:

$$\overrightarrow{ep} \;=\; \langle\, \mathsf{C}\colon \mathsf{swap}(2 : \tau_1^Y,\ 3.9216 : \tau_2^Y) \,\rangle.$$

The swap action is infeasible because the destination pool used for the swap has changed due to the execution of the previous intents: user C requests 3 units of τ_1, but under the updated reserves the constant-product output is only $v = \frac{103 \cdot 2}{194.17 + 2} = \frac{206}{196.17} \approx 1.050 < 3$. Since the minimum-output constraint is violated, the proposal for I_3 is discarded, and the underlying intent is appended to the carry-over order book $\mathcal{I}'' = \mathcal{I}' \circ I_3$ to be retried in a subsequent round, provided it has not expired (i.e., $t_{\mathsf{exp}_{I_3}} > \text{now}$).

For this reason, Let τ_p denote the protocol's reward unit. Consistent with the split outcome (one wins for S_1, one for S_2), credit $\mathsf{S}_1[1 : \tau_p],\ \mathsf{S}_2[1 : \tau_p]$. Moreover, S_1 is the winner of the execution of a cross-chain swap, so it gets the 4 token τ_0 that the user A has burned to perform the cross-

chain. The resulting financial state and configuration transition are:

$$\begin{aligned}\Gamma_2 = \Gamma^\star \;=\; & \mathsf{A}_X[\,6 : \tau_0^X\,] \mid \mathsf{A}_Y[\,12.69 : \tau_1^Y\,] \mid \mathsf{B}_Y[\,16.0092 : \tau_0^Y,\; 7 : \tau_1^Y\,] \mid \\ & \mathsf{C}_Y[\,5 : \tau_1^Y\,] \mid \mathsf{S}_1[\,2 : \tau_p\,] \mid \mathsf{S}_{1X}[\,4 : \tau_0^X\,] \mid \mathsf{S}_2[\,1 : \tau_p\,] \mid \\ & \{\,104 : \tau_0^Y,\; 192.31 : \tau_1^Y\,\} \mid \{\,103 : \tau_1^Y,\; 194.17 : \tau_2^Y\,\} \mid \\ & \{\,105.83 : \tau_2^Y,\; 189.91 : \tau_0^Y\,\}\end{aligned}$$

and the final configuration is:

$$\langle \Gamma_0, \mathcal{I}'', \mathsf{Settle}(\Gamma^\star, \emptyset, \emptyset)\rangle \xrightarrow{\mathsf{perform}} \langle \Gamma^\star, \mathcal{I}'', \mathsf{Inactive}\rangle$$

This yields a *split-winner* batch: S_1 wins I_1, S_2 wins I_2, while I_3 is carried over into the updated order book and is valid for execution up to its expiry (now $< t_{\mathsf{exp}_{I_3}}$).

4.5 Related Work and Discussion

This section compares our formal model against the relevant literature on intent-based protocols in DeFi. In particular, we focus on two intersecting research lines: (i) practical implementations and conceptual models of intent-centric protocols in DeFi, and (ii) auction mechanisms that govern solver competition and influence transaction execution. The first area is closely aligned with the core objectives of our work, while the second represents a complementary domain with considerable potential impact and improvement for our work.

In the first research line, Chitra *et al.* [5] develop both probabilistic and deterministic models to analyze the economic incentives of solvers participating in Dutch auctions. More precisely, the authors examine how entry and congestion costs influence solver participation, effort levels, and resulting user welfare, particularly under conditions of limited market liquidity. While their analysis offers significant microeconomic insight into single-intent scenarios, our work diverges from the abovementioned one in both scope and abstraction. Whereas Dutch auctions execute trades sequentially and provide limited support for coordination, our batch-based model enables the simultaneous settlement of multiple intents, direct matching, and liquidity aggregation. Furthermore,

while Chitra *et al.* focus on incentive structures and entry dynamics, we formalize the full operational semantics of intent-centric protocols using an LTS. Canidio *et al.* [64] introduce a mechanism that selects among multiple fair allocations based on well-defined criteria, without requiring knowledge of users' specific utility functions. They formalize fairness using properties such as envy-freeness and local rationality, and propose an auction framework that satisfies these criteria under combinatorial constraints. In contrast, our work tackles complementary challenges by focusing on the execution semantics and system-level design of intent-based protocols. We model the full lifecycle of intent resolution using an LTS that formally captures the behaviors of solvers, AMM, and batch auctions. In summary, the two papers address a distinct yet complementary dimension: Canidio *et al.* articulate fairness in allocation with combinatorial preferences, while this paper provides a formal execution model for intent-based protocols. Another contribution to the design of fair and MEV-resistant DEX is by McMenamin *et al.* [69]. They propose a protocol based on Width-Sensitive Frequent Batch Auctions (WSFBA) mechanism. This enables market orders without price estimation and ensures timely execution. Their on-chain implementation, FairTraDEX, uses zero-knowledge proofs and escrow-based participation to ensure anonymity, accountability, and fixed execution costs. While McMenamin *et al.* focus on enforcing fairness and privacy through cryptographic techniques within a specific auction-based framework where users submit transactions directly, our work takes a broader, formal systems approach rooted in intent-based protocols, where users submit high-level trade intents to be executed by external solvers. Our LTS models the operational semantics of these protocols, capturing interactions between users, solvers, and settlement layers. Also, our model supports a more flexible execution environment that includes both batch auctions and AMMs and abstracts the execution logic to accommodate a range of mechanisms, offering a more extensible basis for protocol-level reasoning and verification.

Auction mechanisms are a fundamental building block in many DeFi architectures. While our work does not aim to design or optimize these

mechanisms in isolation, we treat them as modular, abstract components within a comprehensive formal framework for modeling intent-based systems. From a theoretical standpoint, several studies have framed the problem of trade execution as a routing optimization challenge. Two notable examples are the works of Angeris *et al.* [33] and Diamandis *et al.* [34] represent two relevant projects that formalize optimal routing in CFMM-based networks to maximize user utility while minimizing costs. Similarly, Danos *et al.* [35] explore arbitrage opportunities across exchange networks, proposing global routing algorithms that exploit price discrepancies. While these works offer valuable insights into how solver strategies can be instantiated, they treat routing as an internal optimization layer, an approach that complements but does not overlap with our contribution. In our model, solver behavior is completely abstracted, focusing on the broader protocol-level dynamics governing intent lifecycle, coordination, and settlement.

4.5.1 Discussion

Intent-based protocols involve complex, asynchronous interactions between users, solvers, and on-chain systems. To obtain a tractable model, we introduced some simplifying assumptions that abstract away some of the protocol's dynamic and unbounded characteristics.

We model the protocol as a sequence of explicit stages regarding the intent collection, the proposal collection/auction, and the settlement phase. This separation allows for a more structured and analyzable representation of the protocol's overall behavior.

First of all, at this stage, the set of solvers is fixed within a round; each solver may submit at most one proposal plan per intent. A solver may also submit an empty plan, which effectively corresponds to not submitting any proposal. Moreover, we abstract time and partial asynchrony via an unbounded queue of intents: intents submitted during the intent collection phase are buffered and become eligible for execution in the subsequent active round. Our model treats intents as atomic, and we do not allow partial execution and decomposition into sub-intents. In con-

trast, many production protocols support partial fills to improve pricing and fill rates. For instance, 1inch Fusion+ allows multiple solvers to execute slices of a large swap; CoW Protocol clears a single order against multiple counterparties within a batch auction; UniswapX permits Dutch orders to be filled in segments over time; and 0x supports partially fillable limit orders. At this stage, we adopt atomicity to keep the formal model clear, leaving a full treatment of partial-fill semantics to future work.

All the intents included in the frozen order book do not expire during the round: the round window is chosen so that its deadline is strictly earlier than the minimum deadline among the included intents, preventing mid-round expiration. During settlement, if a selected proposal fails, the system discards it entirely and does not attempt to execute any partial plan derived from it. Finally, we assumed that each agent plays a single role as a user or as a solver. Weakening this assumption may introduce *conflicts of interests*, during the auction and the execution of intents, where a solver that is also a user may be incentivized to prioritize their own intents over others. This may lead to unfair execution, front-running, or inefficient batching. These behaviors threaten the integrity of the protocol and emphasize the need for strong mechanisms to prevent manipulative strategies. By enforcing a strict separation between users and solvers, we simplify the strategic environment and eliminate opportunities for cross-role optimization that could distort outcomes. Nevertheless, future extensions of the model may explore safe mechanisms that allow for dual participation while effectively mitigating the associated conflicts.

Directly Matchable Intents

CoW Protocol introduces the notion of *Coincidence of Wants*: when two users each hold exactly what the other wants, their orders can be matched peer-to-peer and settled as a direct barter. The matched portion bypasses external AMM liquidity and its LP fees; users also do not pay gas directly, as settlement is handled through CoW's contracts. CoWs may be full or partial: in the former version of CoW, both orders are entirely matched against each other; in the latter version of CoW, only part of the volume

matches, and a solver supplies the remaining liquidity to complete the batch. Formally, a full CoW occurs when two intents

$$I = \mathsf{U_i} : (a, \tau_x, b, \tau_y, t_{\mathsf{exp}}) \quad \text{and} \quad I' = \mathsf{U_j} : (b, \tau_y, a, \tau_x, t'_{\mathsf{exp}})$$

are *directly matchable*, written $I \leftrightarrows I'$, i.e., they are exact reversals in token and amount and both are unexpired:

$$I \leftrightarrows I' \iff (a : \tau_x \text{ of } I_{\mathsf{U_i}} = a : \tau_x \text{ of } I'_{\mathsf{U_j}} \wedge b : \tau_y \text{ of } I_{\mathsf{U_i}} = b : \tau_y \text{ of } I'_{\mathsf{U_j}} \wedge \ now < t_{\mathsf{exp}} \ \wedge \ now < t'_{\mathsf{exp}}), \tag{4.6}$$

In a partial CoW, the reversal holds but not necessarily for the full size: the two intents match on their overlapping volume, and any residual imbalance is sourced by solver-provided liquidity or order aggregation within the same batch. To support this mechanism in our system, we must relax the per-intent execution model: instead of restricting each solver to a plan tied to a single intent, during Active_Round, solvers should be allowed to propose multi-intent plans that treat all the intents of the frozen snapshot $\mathcal{I}$ (for instance, combining direct matches with auxiliary AMM trades). We therefore introduce a primitive action $\mathsf{exchange}(I\,,\,I')$ indicating that a pair of intents $I, I' \in \mathcal{I}$ can be directly matched and fulfilled without using AMM liquidity; this is settled by the [P2P_EXCHANGE] rule. The execution of a direct match is then managed as follows:

[P2P_EXCHANGE]

$$\frac{I = \mathsf{A} : (a, \tau_x, b, \tau_y, t_{\mathsf{exp}}) \quad I' = \mathsf{A}' : (b, \tau_y, a, \tau_x, t'_{\mathsf{exp}}) \quad I \leftrightarrows I' \quad \Gamma' = \mathsf{A}[\sigma - a : \tau_x + b : \tau_y] | \mathsf{A}'[\sigma' + a : \tau_x - b : \tau_y]}{\mathsf{A}[\sigma] | \mathsf{A}'[\sigma'] \mid \Gamma'' \xrightarrow{\mathsf{exchange}(I\,,\,I')} \Gamma' \mid \Gamma''}$$

In the premise of the rule, we check that the two intents I and I' are directly matchable according to equation (4.6), and we implicitly require that involve users A and A' whose wallets are in the current financial state. If these checks hold, we immediately update users' wallets as requested by the intents with no on-chain operation. The resulting state reflects this update.

Cross-chain Swap

The auxiliary system supports swaps and cross-chain swaps. To model cross-chain exchanges, the financial state is parameterized by chain identifiers: balances, user wallets, and liquidity reserves are all indexed by chain, and a user may control distinct wallets on multiple chains. The cross-chain big-step rule we introduce encapsulates the whole operation as one atomic performance step, even though the underlying mechanism is more complex.

In detail, a cross-chain swap consists of the following steps: first of all, the user burns the source asset on chain X and opens a cross-chain order under a new identifier *id*, establishing supply neutrality and anti-replay. Following this, a solver uses its own liquidity to execute the destination chain Y and delivers the output asset to the user; and finally, the solver is repaid on the source chain X with the burned amount, contingent on a valid receipt from the destination chain Y.

In this chapter, we focus on the intent-based protocol and its operational labels. To keep the scope tight, we do not introduce an auxiliary small-step transition system for cross-chain execution: the cross-chain swap is modeled as a single big-step rule rather than decomposed into its micro-steps (burn, deliver, settle). An in-depth formalization of the cross-chain layer, including multi-chain state synchronization, per-chain wallet accounting, chain-indexed reserves, and anti-replay guarantees, is addressed in future work.

Chapter 5

Runtime Verification for History-Based Enforcement

Smart contracts are computer programs deployed and executed within a blockchain environment. On the Ethereum[1]blockchain, they are commonly written in the Solidity language and compiled into the EVM bytecode. The definition of a smart contract in Solidity looks like a class in any OOP language: contracts have an internal mutable state and a set of functions to manipulate it. These public functions can be invoked by users directly through transactions or other contracts through the external calls mechanism. Although this mechanism is powerful in enabling interactions between smart contracts, it provides no means to ensure that the invoked code satisfies some predefined behavioral policies. This is even more critical when we pass the address of a smart contract as an argument to a function that, in turn, invokes a function within the passed contract, so letting the caller control the code that is actually executed. For example, a contract `Bank` may implement a function `calculateInterest(RateProvider pro)` by invoking the function `getCurrentRate()` on the parameter `pro` of type `RateProvider` to

[1]Ethereum was designed from the ground up with a quasi Turing-complete language. While the language itself is Turing complete, computations are associated with a bounded (called gas), which gets consumed by each instruction thereby enforcing termination [70].

obtain the current rate of a given crypto-asset in exchange for some fees. The trustworthiness of the computation depends on the code of `getCurrentRate()`. Relying on external code could have severe security consequences as many attacks on smart contracts exploit external calls to run attacker-controlled code, e.g., reentrancy attacks [71, 72].

To address this issue, we propose a methodology to specify and enforce security policies at contract code level, inspired by Schneider [73] and Kozen [74]. The methodology involves (i) extending the programming language with constructs for expressing security policies to guard pieces of code; and (ii) introducing inside the semantics of the language mechanisms to check that the code is compliant with such policies at run-time.

More precisely, we start from TinySol [75], a core calculus for smart contracts, and we extend it with a policy framing construct $\phi[S]$ for guarding the execution of statements S with a developer-defined policy ϕ. Intuitively, a policy ϕ is a predicate that specifies the set of program executions that are acceptable. The predicate ϕ is a pair (re, E), where re is a regular expression and E is an assertion (boolean expression). The regular expression re predicates on the *execution history*, namely the sequence of function calls made by the contract. While the expression E predicates on the state of the computation, i.e., the values of contract and function variables. Since the values of these variables will be only known at run-time, we allow them to occur also inside re. These variables will be bound to concrete values while evaluating re against the execution history. We define the semantics of TinySol as a transition system that describes the result of a computation (the reaching state), and that keeps track of the sequence of function calls performed at run-time together with their actual parameters (the *history*). Moreover, we define a run-time procedure to check when the execution satisfies the policies of a contract. This procedure works as follows: given a policy $\phi = (re, E)$, we first obtain a symbolic automaton from re that accepts all the histories that may be compliant for some assignment of variables occurring in re; then, we evaluate the assertion E to check its validity. A history is compliant if it passes both these checks, otherwise it is not, and the execu-

tion is reverted. This chapter addresses these challenges by contributing a formal policy framework aimed at specifying and enforcing security and correctness constraints in smart contracts. Our work underscores the importance of developing an automated, specialized approach that bridges the gap between formal analysis and real-world applicability, ultimately supporting a more secure and resilient DeFi ecosystem. The content of the chapter is based on [76] and is organized as follows: Section 5.1 provides some background information and motivations for our investigation. Section 5.2 presents the formalization of our methodology. Section 5.3 shows our use cases on logic and application vulnerabilities. Section 5.4 compares our work with the relevant literature.

5.1 Context and Motivation

The emergence of Ethereum and other blockchain platforms supporting smart contracts has significantly reshaped the digital landscape. By enabling the creation of decentralized applications (dApps) that operate without trusted intermediaries, these technologies have laid the foundation for a new wave of financial innovation. Among the most transformative developments is DeFi, a fast-growing ecosystem that offers financial services such as lending, trading, and investment through programmable smart contracts. Since 2020, DeFi has experienced unprecedented growth, with the TVL reaching 115.5 billion USD as of July 2025.

DeFi represents an innovative financial paradigm built on blockchain infrastructures, where autonomy, composability, and transparency are key features. While Ethereum remains the dominant platform, DeFi ecosystems have also expanded to other networks such as Solana. However, the ecosystem remains highly fragmented. Prices for identical assets can vary significantly across venues, and these venues often exhibit inconsistent responses to market dynamics. This fragmentation introduces inefficiencies and opens the door to economically motivated attacks, such as price manipulation and oracle exploitation, which can result in either unfair advantages for attackers or significant losses for honest participants.

The very qualities that make DeFi appealing - its openness, interop-

erability, and decentralization - also contribute to its vulnerability. Despite growing awareness and improved development practices, attackers continue to exploit flaws in smart contract logic. According to the FailSafe Web3 Security Report [77], DeFi has suffered over 2.6 billion USD in losses across 192 major incidents, highlighting the urgent need for robust and formal security mechanisms.

Smart contracts serve as the foundation of DeFi applications, where they implement financial logic directly on-chain. Their flexibility enables complex and expressive financial interactions but also introduces serious challenges in ensuring correctness and security. These challenges are particularly critical in adversarial environments that involve liquidity constraints, arbitrage, reentrancy, or cross-protocol interactions. Traditional methods, such as manual auditing and unit testing, are insufficient to guarantee the absence of critical bugs or vulnerabilities. Instead, rigorous formal methods are essential for modeling, verifying, and enforcing security properties at the code level. Consistent with recent findings by Chaliasos *et al.* [78], we emphasize that current automated tools are limited in scope. While they may effectively detect well-known issues like reentrancy, they fall short in identifying deeper, logic-related bugs and protocol-layer vulnerabilities - including oracle manipulation and cross-market price manipulation attacks. Practitioners increasingly recognize these categories as some of the most impactful and least addressed threats. Moreover, the usability and precision of current tools often fail to meet the practical demands of smart contract developers and auditors. Here, we propose history-based policies as a means to regulate the composition of various protocols, thereby enforcing good behavior among the involved parties. More precisely, our policy framework addresses security concerns by analyzing transaction patterns and protocol interactions to identify potential vulnerabilities and proactively strengthen defenses against malicious activities. Among these, reentrancy attacks are particularly dangerous due to their recursive nature, allowing malicious actors to repeatedly invoke a vulnerable function before the contract's internal state is updated.

To address these limitations, we propose a *history-based policy frame-*

work that governs the composition of protocols by analyzing transaction sequences and behavioral patterns over time. Rather than reasoning about contracts in isolation, our approach monitors system-level interactions to detect anomalous flows and enforce intended constraints dynamically. In particular, protecting the invocation of sensitive functions (such as withdrawals) with a runtime policy ϕ allows us to detect and block profitable but undesirable behaviors, such as recursive exploits, as they unfold. Such runtime monitoring provides a flexible and expressive mechanism for enforcing behavioral properties over execution traces, augmenting static analysis with real-time defense.

One notable example of a reentrancy vulnerability is the DAO exploit [79, 80]. In June 2016, attackers exploited a flaw in the DAO smart contract, which allowed users to withdraw funds via a split function. Due to incorrect ordering of operations, which transfers Ether before updating the user's balance, a malicious contract could recursively trigger multiple withdrawals before the internal state was updated. This enabled the attacker to drain over 3.6 million ETH (worth more than 60 million USD at the time). The attack caused a major crisis in the Ethereum community and ultimately led to a hard fork of the blockchain.

To better illustrate how reentrancy vulnerabilities can be exploited, we introduce a representative case: the mono-function reentrancy attack. In this scenario, an attacker exploits a flawed withdrawal logic to recursively drain funds from a smart contract. This attack involves two contracts: a vulnerable contract and a malicious one. As shown in Figure 18, we define two simplified contracts, `ChainWallet` (the vulnerable contract) and `ChainReentrancy` (the attacker contract), to demonstrate the attack pattern.

The attack begins when several users (Alice, Bob, and Kate) deposit Ether into `ChainWallet`. The attacker then deploys `ChainReentrancy` and performs a minimal deposit before invoking its `attack()` function. This triggers a call to the flawed `withdraw()` function in `ChainWallet`, which sends Ether to the caller using a low-level call before updating the caller's balance. Because the caller is a contract (`ChainReentrancy`), its `receive()` fallback function is triggered, allowing it to re-enter `withdraw()`

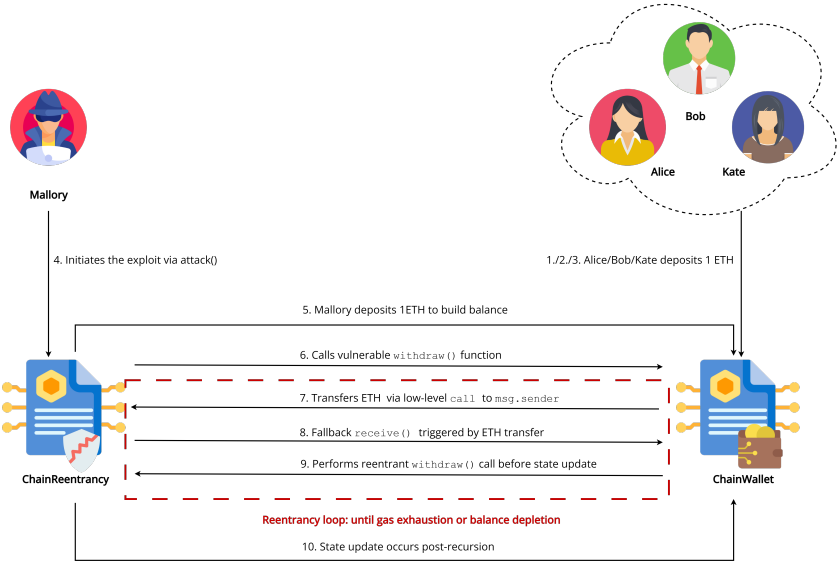

Figure 18: Sequence diagram illustrating the mono-function reentrancy attack.

recursively. The balance remains unchanged during this loop, enabling the attacker to drain the funds.

The vulnerable `withdraw()` function is shown in the following Solidity snippet:

Listing 5.1: Vulnerable `withdraw` function.

```
mapping(address => uint) public balances;

function withdraw() public {
    // Checks user's balance
    require(balances[msg.sender] > 0, "Insufficient balance");

    // Sends all the user's balance to them
    (bool success, ) = msg.sender.call{value: balances[msg.sender]}("");
        //  At this point, the caller's code is executed
    require(success, "Failed to send ETHs");

    // Updates the user's balance after sending them their entire balance
    balances[msg.sender] = 0; // State update occurs too late
}
```

Because the Ether transfer occurs prior to resetting the user's balance, the attacker's contract can invoke `withdraw()` recursively via its `receive()` function. This cycle continues until the contract's balance is fully drained.

This vulnerability highlights the importance of adhering to the checks-effects-interactions pattern, where internal state changes are applied before any external calls are made. Reentrancy attacks, as part of the broader category of transaction-level exploits, underscore the necessity for rigorous smart contract auditing, formal verification, and runtime protections. When combined with our proposed history-based policy framework and runtime monitoring via policies like ϕ, the system gains a behavior-aware, context-sensitive enforcement layer that proactively detects and reacts to abnormal patterns indicative of exploitation.

5.2 TinySol with Policies

In this section, we introduce the core calculus for smart contracts, building upon the work of Bartoletti *et al.* [75]. First, we present the syntax and semantics of the enriched language, with a particular focus on the introduced policy component (Sections 5.2.1 and 5.2.2). Following this, we formalize the execution of transactions using a labeled transition system (LTS). Finally, we review symbolic automata and show how we use them to define our policy checking mechanism (Section 5.2.3).

5.2.1 Syntax

We assume a set **Val** of *values* ranged by $v, k, \ldots$, a set **Const** of *constant names* $x, y, \ldots$, a set of *procedure names* $\mathtt{f}, \mathtt{g}, \ldots$. and a set **Addr** of *addresses* $\mathcal{X}, \mathcal{Y}, \ldots$, partitioned into *account addresses* $\mathcal{A}, \mathcal{B}, \ldots$ and *contract addresses* $\mathcal{C}, \mathcal{D}, \ldots$. As a notation, we write sequences in bold, e.g. $\boldsymbol{v}$ is a sequence of values, while ϵ denotes the empty sequence. We use $n, n', \ldots$ to range over $\mathbb{N}$, and $b, b', \ldots$ to range over boolean values.

In TinySol, a *contract* is a finite set of terms of the form $\mathtt{f}(\boldsymbol{x})\{S\}$, where S is a *statement*. Each term $\mathtt{f}(\boldsymbol{x})\{S\}$ is a contract function, where $\mathtt{f}$ is its

name, $\boldsymbol{x}$ are its formal parameters (omitted when empty), and S is the function body. Moreover, we represent the internal state of a contract as a key-value store, formally, a partial function from keys $k \in \mathbf{Val}$ to values $v \in \mathbf{Val}$.

The syntax of TinySol is in Figure 19. Many statements (Figure 19, left) are standard in common imperative languages, so we only comment on the most relevant: `throw` raises an uncatchable exception, rolling-back the state; $\mathfrak{X}.\mathtt{f}(\boldsymbol{v})\$n$ implements the external call mechanism calling the function `f` of the contract at address $\mathfrak{X}$, passing the arguments $\boldsymbol{v}$, and transferring n units of currency to $\mathfrak{X}$; $\phi[S]$ executes the statement S guarded by the policy ϕ, aborting the execution if the history generated by S is not compliant with ϕ.

Also, the expressions (Figure 19, right) are quite standard. We assume all the usual constants, arithmetic, logic, and cryptographic operators. Peculiar of TinySol is the expression $!k$ that evaluates to true if the key k is bound in the contract store, false otherwise. Then, the expression $?k$ looks up the value of the key k in the contract store. Finally, the expression $E_1.E_2$ evaluates E_1 to produce the address of a contract $\mathfrak{X}$ and then evalutes E_2 in that address.

A *policy* ϕ (Figure 19, left) is a pair (re, E) where re is a regular expression that represents the sequence of function invocations, and E is a boolean expression. The regular expression syntax is standard: $\cdot$ denotes concatenation, $+$ choices, and * zero or more repetitions. The alphabet is represented by patterns patt capturing function invocation and may contain variables. More precisely, $\mathfrak{C}.\mathtt{f}(\boldsymbol{v})$ denotes an invocation to the function `f` of the contract with address $\mathfrak{C}$ with argument $\boldsymbol{v}$; whereas $x.\mathtt{f}(\boldsymbol{v})$ is a call to function `f` with argument $\boldsymbol{v}$ where x is a placeholder for the address of the contract the function belongs to; $\mathfrak{C}.\mathtt{f}(\boldsymbol{x})$ denotes a call where the arguments are variables; finally, $x.\mathtt{f}(\boldsymbol{x'})$ is a call where the address and the arguments are denoted by variables. The variables occurring in a pattern can be used by the expression E to express some conditions. These variables will be bound to concrete values while checking the policy compliance.

Finally, we assume a mapping Γ from addresses to contracts that

we use to retrieve the definition of a contract. In order to allow for a uniform treatment of account and contract addresses, we require that $\Gamma(\mathcal{A}) = \{\mathtt{f}_{\mathtt{skip}}()\{\mathtt{skip}\}\}$ for all account addresses $\mathcal{A}$. Also, we use the following syntactic sugar: we write $\mathcal{X}.\mathtt{f}(\boldsymbol{v})$ to denote a call $\mathcal{X}.\mathtt{f}(\boldsymbol{v})\$n$, when there is no money transfer (i.e., $n = 0$). We write $\mathtt{if}\ E\ \mathtt{then}\ S$ for $\mathtt{if}\ E\ \mathtt{then}\ S\ \mathtt{else}\ \mathtt{skip}$.

$S ::=$	statement	$E ::=$	expression
$\mathtt{skip}$	skip	v	value
$\mid \mathtt{throw}$	exception	$\mid x$	const name
$\mid E := E'$	store update	$\mid \mathcal{X}$	address
$\mid S; S'$	sequence	$\mid \mathtt{op}\ \boldsymbol{E}$	operator
$\mid \mathtt{if}\ E\ \mathtt{then}\ S\ \mathtt{else}\ S'$	conditional	$\mid ?E$	key lookup
$\mid \mathtt{while}\ E\ \mathtt{do}\ S$	loop	$\mid !E$	key bound?
$\mid \phi[S]$	security policy	$\mid E_1.E_2$	context
$\mid E_0.\mathtt{f}(\boldsymbol{E_1})\$E_2$	call		
		$patt ::=$	function calls
$\phi ::= (re, E)$	policy	$\mathcal{C}.\mathtt{f}(\boldsymbol{v})$	ground
$re ::=$	regular expression	$\mid x.\mathtt{f}(\boldsymbol{v})$	addr. variable
$patt$	basic event	$\mid \mathcal{C}.\mathtt{f}(\boldsymbol{x})$	arg. variables
$\mid \epsilon$	empty string	$\mid x.\mathtt{f}(\boldsymbol{x'})$	addr. and arg. variables
$\mid re \cdot re$	sequential composition		
$\mid re + re$	choice		
$\mid re^*$	iteration		

Figure 19: Syntax of the language.

5.2.2 Semantics

We formalize the execution of TinySol contracts by providing operational semantics for both transactions and function contracts. In our semantics, the execution of a transaction is nondeterministically started by a user account, and it triggers the execution of a contract function. A successful transaction results in a change of the blockchain state. We first present the semantics of TinySol statements and then the semantics of transactions. Note that our policy mechanism is activated during statement execution but affects transaction semantics since the corresponding transaction is reverted when a policy is violated.

Statements

The semantics of a contract is given in terms of a transition system where configurations have the form:

$$\langle S, \sigma, \rho, \eta \rangle$$

where S is the sequence of statements to be executed; σ is the starting state; ρ is the variable environment; η is the history recording the sequence of function calls, made during the execution.

In detail, a *blockchain state* $\sigma : \mathbf{Addr} \to (\mathbf{Val} \rightharpoonup \mathbf{Val})$ is a map from addresses to key-value stores. A key-value store is, in turn, a partial function from keys to values and represents the internal state of a smart contract. We use brackets to represent finite maps, e.g., $\{{}^{v_1}/_{x_1}, \cdots, {}^{v_n}/_{x_n}\}$ maps x_i to v_i, for $i \in 1..n$. As usual, when a key k is not bound to any value in $\sigma\, \mathcal{X}$, we write $\sigma\, \mathcal{X}\, k = \bot$. We postulate that for each address, there exists the distinct key `balance` recording the balance of that address. Moreover, when we refer to the key of an address $\mathcal{X}$, we use the notation $\mathcal{X}.k$ and call it a qualified key. Consequently, we write $\sigma(\mathcal{X}.k)$ for $\sigma \mathcal{X} k$. Below, we use the auxiliary operators $\sigma + \mathcal{X} : n$ and $\sigma - \mathcal{X} : n$ on states, to, respectively, increase/decrease the `balance` of $\mathcal{X}$ of n currency units:

$$\sigma \circ \mathcal{X} : n \;=\; \sigma\{{}^{(\sigma\, \mathcal{X}\, \texttt{balance}) \circ n}/_{\mathcal{X}.\texttt{balance}}\} \qquad (\circ \in \{+, -\})$$

A variable environment $\rho : \mathbf{Const} \rightharpoonup \mathbf{Val}$ is a partial function from constant name to values used to provide values to function parameters (that once assigned cannot be changed) and to special names such as `value` and `sender` used inside the body of functions.

A history η is a sequence of function calls given by the following grammar:

$$\eta ::= \epsilon \mid \eta \cdot \eta \mid \mathcal{X}.\texttt{f}(\boldsymbol{v})$$

where ϵ is the empty history, $\mathcal{X}.\texttt{f}(\boldsymbol{v})$ denotes the invocation of the function `f` of the contract with address $\mathcal{X}$ passing the values $\boldsymbol{v}$ as arguments.

We give the operational semantics of statements in a big-step style, where transitions have the form:

$$\langle S, \sigma, \rho, \eta \rangle \xrightarrow{\mathcal{X}} \langle \sigma', \eta' \rangle$$

meaning that the statement S executed in the contract $\mathcal{X}$, in the state σ, with environment ρ, and history η terminates and produces the final state σ' and history η'. Note that our semantics is parameterized by the contract address in which the evaluation is taking place.

Figure 20 shows the semantics of expressions where we use `op` to denote syntactic operators and op for their semantic counterparts. The semantics of expression results in a value and its definition is quite standard: the environment ρ is used to evaluate constant names x, while the state σ is used to evaluate keys in $!E$ and $?E$. The semantics of $E_1.E_2$ first evaluates E_1 within a contract with address $\mathcal{X}$ producing an address $\mathcal{Y}$, then evaluates E_2 within the contract $\mathcal{Y}$. Note that expressions have no side effects, and all the semantic operators are *strict*, i.e. their result is $\perp$ if some operand is $\perp$.

The semantics of statements is in Figure 21, and it is mostly standard, except for the last two rules. The rule [POLICY] evaluates the policy framing $\phi[S]$ construct: it executes the statement S that (if it terminates) produces a state σ' and a history η'; then, it checks that η' is compliant with the policy ϕ through the predicate $\eta' \vDash_{\sigma',\rho,\mathcal{X}} \phi$ defined in Section 5.2.3. If the history meets the policy, the computation results in the pair $\langle \sigma', \eta' \rangle$, otherwise it is aborted.

The rule [PROCEDURE CALL] handles a procedure call $E_0.\mathtt{f}(\boldsymbol{E_1})\$E_2$ within the contract $\mathcal{X}$. The premise of the rule requires that (i) E_0 evaluates to an address $\mathcal{Y}$; (ii) E_2 evaluates to a non-negative number n, not exceeding the balance of $\mathcal{X}$; (iii) the contract at $\mathcal{Y}$ has a procedure named `f` with formal parameters $x_1 \cdots x_h$; (iv) $\boldsymbol{E_1}$ evaluates to a sequence of values of length h. If all these conditions hold, then the procedure body S is executed in a state where $\mathcal{X}$'s balance is decreased by n, $\mathcal{Y}$'s balance is increased by n, and in an environment where the formal parameters are bound to the actual ones, and the special names `sender` and `value` are bound, respectively, to $\mathcal{X}$ (the caller) and n (the value transferred to $\mathcal{Y}$). Executing S may affect both the store of $\mathcal{X}$ and, in case of procedure calls, also the store of other contracts.

$$[\![v]\!]^{\mathfrak{X}}_{\sigma,\rho} = v \qquad [\![x]\!]^{\mathfrak{X}}_{\sigma,\rho} = \rho\, x \qquad [\![\mathcal{Y}]\!]^{\mathfrak{X}}_{\sigma,\rho} = \mathcal{Y} \qquad [\![\mathtt{op}\,\boldsymbol{E}]\!]^{\mathfrak{X}}_{\sigma,\rho} = \mathtt{op}\,[\![\boldsymbol{E}]\!]^{\mathfrak{X}}_{\sigma,\rho} \qquad [\![E_1.E_2]\!]^{\mathfrak{X}}_{\sigma,\rho} = [\![E_2]\!]^{[\![E_1]\!]^{\mathfrak{X}}_{\sigma,\rho}}_{\sigma,\rho}$$

$$[\![?\,E]\!]^{\mathfrak{X}}_{\sigma,\rho} = \sigma\,\mathfrak{X}\,([\![E]\!]^{\mathfrak{X}}_{\sigma,\rho}) \qquad [\![!\,E]\!]^{\mathfrak{X}}_{\sigma,\rho} = \begin{cases} true & \text{if } [\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} \neq \bot \text{ and } \sigma\,\mathfrak{X}\,([\![E]\!]^{\mathfrak{X}}_{\sigma,\rho}) \neq \bot \\ false & \text{if } [\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} \neq \bot \text{ and } \sigma\,\mathfrak{X}\,([\![E]\!]^{\mathfrak{X}}_{\sigma,\rho}) = \bot \end{cases}$$

Figure 20: Semantics of expressions.

$$\frac{}{\langle \mathtt{skip}, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma, \eta\rangle}\,[\textsc{skip}] \qquad \frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = k \quad [\![E']\!]^{\mathfrak{X}}_{\sigma,\rho} = v}{\langle E{:=}E', \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma\{{}^{v}\!/_{\mathfrak{X}.k}\}, \eta'\rangle}\,[\textsc{update}]$$

$$\frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = b \quad b \in \{true, false\}}{\langle \mathtt{if}\ E\ \mathtt{then}\ S_{true}\ \mathtt{else}\ S_{false}, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle S_b, \eta'\rangle}\,[\textsc{condition}] \qquad \frac{\langle S_0, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}{\langle S_0; S_1, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\,[\textsc{sequence}]$$

$$\frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = false}{\langle \mathtt{while}\ E\ \mathtt{do}\ S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma, \eta\rangle}\,[\textsc{while false}] \qquad \frac{[\![E]\!]^{\mathfrak{X}}_{\sigma,\rho} = true \quad \langle S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}{\langle \mathtt{while}\ E\ \mathtt{do}\ S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\,[\textsc{while true}]$$

$$\frac{\langle S, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle \quad \eta' \vDash_{\sigma',\rho,\mathfrak{X}} \phi}{\langle \phi[S], \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma', \eta'\rangle}\,[\textsc{policy}]$$

$$\frac{\begin{array}{lll} [\![E_0]\!]^{\mathfrak{X}}_{\sigma,\rho} = \mathcal{Y} & \mathtt{f}(x)\{S\} \in \Gamma(\mathcal{Y}) & \eta' = \eta \cdot \mathcal{Y}.f(v) \\ [\![\boldsymbol{E_1}]\!]^{\mathfrak{X}}_{\sigma,\rho} = \boldsymbol{v} & \sigma' = \sigma - \mathfrak{X} : n + \mathcal{Y} : n & \langle S, \sigma', \rho', \eta'\rangle \xrightarrow{\mathcal{Y}} \langle \sigma'', \eta''\rangle \\ [\![E_2]\!]^{\mathfrak{X}}_{\sigma,\rho} = n & \rho' = \{{}^{\mathfrak{X}}\!/_{\mathtt{sender}}, {}^{n}\!/_{\mathtt{value}}, {}^{v}\!/_{x}\} & \end{array}}{\langle E_0.\mathtt{f}(\boldsymbol{E_1})\$E_2, \sigma, \rho, \eta\rangle \xrightarrow{\mathfrak{X}} \langle \sigma'', \eta''\rangle}\,[\textsc{procedure call}]$$

Figure 21: Semantics of statements.

Transactions

A *transaction* T is a term of the form $\mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})$, where $\mathcal{A}$ is the address of the caller, $\mathcal{C}$ is the address of the called contract, $\mathtt{f}$ is the called procedure, n is the value transferred from $\mathcal{A}$ to $\mathcal{C}$, and $\boldsymbol{v}$ is the sequence of actual parameters. We formalize the execution of a transaction T through a transition system where configurations have the form $\langle \sigma, \eta\rangle$ describing the blockchain state where T takes place producing a new configuration $\langle \sigma', \eta'\rangle$. Executing the transaction causes the execution of the contract function $\mathtt{f}$. The call of $\mathtt{f}$ is defined by the following rules:

$$\frac{\mathtt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \quad \sigma\,\mathcal{A}\,\mathtt{balance} \geq n \qquad \langle S, \sigma - \mathcal{A} : n + \mathcal{C} : n,\ \{{}^{\mathcal{A}}\!/_{\mathtt{sender}}, {}^{n}\!/_{\mathtt{value}}, {}^{\boldsymbol{v}}\!/_{\boldsymbol{x}}\}, \eta\rangle \xrightarrow{\mathcal{C}} \langle \sigma', \eta'\rangle}{\langle \sigma, \eta\rangle \xrightarrow{\mathcal{A} \xrightarrow{n} \mathcal{C}:\mathtt{f}(\boldsymbol{v})} \langle \sigma', \eta'\rangle}\,[\textsc{Tx1}]$$

$$\frac{\mathtt{f}(\boldsymbol{x})\{S\} \in \Gamma(\mathcal{C}) \qquad \big(\sigma\, \mathcal{A}\, \mathtt{balance} < n \quad \text{or} \quad \langle S, \sigma - \mathcal{A} : n + \mathcal{C} : n,\ \{{}^{\mathcal{A}}/_{\mathtt{sender}}, {}^{n}/_{\mathtt{value}}, {}^{\boldsymbol{v}}/_{\boldsymbol{x}}\}, \eta\rangle \not\to \big)}{\langle \sigma, \eta\rangle \xrightarrow{\mathcal{A} \xrightarrow{n} \mathcal{C} : \mathtt{f}(\boldsymbol{v})} \langle \sigma, \eta\rangle} \text{[Tx2]}$$

Rule [Tx1] handles the case where the transaction is successful: this happens when $\mathcal{A}$'s balance is at least n, and the procedure call terminates in a non-error state, meaning that all policies defined in `f` are satisfied. Note that n units of currency are transferred to $\mathcal{C}$ *before* starting to execute `f`, and that the names `sender` and `value` are set, respectively, to $\mathcal{A}$ and n. Instead, [Tx2] applies either when $\mathcal{A}$'s balance is not enough, or the execution of `f` fails (this also covers the case when `f` does not terminate or during the execution of the function body is not compliant with all the policies of function `f`). In these cases, T does not alter the state $\langle \sigma, \eta\rangle \xrightarrow{\mathsf{T}} \langle \sigma, \eta\rangle$.

We naturally extend the semantics above when we have a sequence of transactions $[\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n]$, formally:

$$\langle \sigma_0, \eta_0\rangle \xrightarrow{\mathsf{T}_1} \langle \sigma_1, \eta_1\rangle \xrightarrow{\mathsf{T}_2} \langle \sigma_2, \eta_2\rangle \cdots \xrightarrow{\mathsf{T}_n} \langle \sigma_n, \eta_n\rangle$$

Note that erroneous transactions occur in the execution of the sequence $[\mathsf{T}_1, \mathsf{T}_2, \cdots, \mathsf{T}_n]$ does not effect the semantics, since rule [Tx2] leaves the state and the history unchanged.

5.2.3 Checking policies

Here, we describe how we ensure that a history η complies with a policy $\phi = (re, E)$. Intuitively, our verification procedure consists of two steps: (i) we check that η belongs to the language of re, by transforming re into a symbolic automaton; (ii) we evaluate the assertion E in the current state and check that it evaluates to true. If both steps succeed, η complies with ϕ, otherwise, it does not. Below, we first review the basic notions of symbolic regular expressions and automata [81], and then we detail the two steps above.

Symbolic automata are generalizations of finite automata where the alphabet is a Boolean algebra, and transitions are labeled by predicates

over such algebra. This allows symbolic automata to operate over infinite alphabets, such as the set of rational numbers, while retaining the decidability properties of their finite counterparts. Similarly, symbolic regular expressions are a generalization of regular expressions operating over an effective Boolean algebra.

An *effective Boolean algebra* $\mathcal{A}$ is a tuple

$$(\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg) \tag{5.1}$$

where $\mathfrak{D}$ is a (possibly infinite) set of *domain elements*; Ψ is a set of *predicates* over $\mathfrak{D}$ closed under the logical connectives $\vee, \wedge, \neg$; the component $[\![\_]\!] : \Psi \to 2^{\mathfrak{D}}$ is a *denotation function* such that $[\![\top]\!] = \mathfrak{D}$, $[\![\bot]\!] = \emptyset$, and $\forall \psi_1, \psi_2 \in \Psi, [\![\psi_1 \vee \psi_2]\!] = [\![\psi_1]\!] \cup [\![\psi_2]\!]$, and $[\![\neg\psi_1]\!] = \mathfrak{D} \setminus [\![\psi_1]\!]$. Moreover, when $[\![\psi_1]\!] \neq \emptyset$, then the predicate ψ_1 is *satisfiable*. We require that checking satisfiability is decidable.

Example 7 (Linear Integer Arithmetic)**.** *As an example of effective Boolean algebra, consider* $(\mathbb{Z}, \Psi, [\![\cdot]\!], \bot, \top, \vee, \wedge, \neg)$*, where* $\mathbb{Z}$ *is the set of integer numbers. The set* Ψ *contains all quantifier-free formulas on integer linear arithmetic, namely, formulas such as* $\psi_{>0}(x) \triangleq x > 0$ *and* $\psi_{odd}(x) \triangleq x\%2 = 1$*. Given a formula* ψ*,* $[\![\psi]\!]$ *denotes the set of integer satisfying it;* $\bot, \top, \vee, \wedge, \neg$ *represent the always false, true predicates and the standard logical connectives. It is noteworthy that satisfiability is decidable for such a Boolean algebra. Indeed, given a formula* ψ*, the interpretation function* $[\![\psi]\!]$ *can use an SMT solver for checking the satisfiability and model generation. For example,* $[\![\psi_{>0}(x)]\!]$ *is true if there exists an integer* x *such that* $x > 0$*, and* $[\![\psi_{odd}(x)]\!]$ *is true if there exists an integer* x *such that* $x\%2 = 1$*. Boolean operations allow for the combination and manipulation of these predicates within algebra.*

Given an effective Boolean algebra $(\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg)$, a *symbolic regular expression* is defined as follows:

- The constants ε and $\emptyset$ are symbolic regular expressions denoting the languages $\{\varepsilon\}$ and $\emptyset$, respectively.
- Any predicate $\psi \in \Psi$ is a symbolic regular expression that accepts the language defined as $\mathcal{L}(\psi) = [\![\psi]\!]$.
- Given symbolic regular expressions sre_1 and sre_2, the expressions $sre_1 + sre_2$, $sre_1 \cdot sre_2$, and sre_1^* are symbolic regular expressions.

Elements of $\mathfrak{D}$ are *characters*. A *word* over $\mathfrak{D}$ is a sequence $a_1 \dots a_m$, where $a_i \in \mathfrak{D}$, $i = 1, \dots, m$. If $m = 0$, then we have the *empty* word, denoted by ε. The set of all words over $\mathfrak{D}$ is denoted by $\mathfrak{D}^\star$. Any language defined by a symbolic regular expression is a *symbolic regular language*.

A *symbolic nondeterministic finite automaton with epsilon transitions*, *s-ϵNFA* is a quintuple

$$\mathcal{N} = (\mathcal{A}, Q, \Delta, I, F)$$

where the alphabet $\mathcal{A}$ is an effective Boolean algebra; Q is a finite set of *states*; $\Delta \subseteq Q \times (\Psi \cup \{\epsilon\}) \times Q$ is a finite set of *transitions*; $I \subseteq Q$ is the set of *initial states*; and $F \subseteq Q$ is the set of *final states*. The language of $\mathcal{N}$, denoted by $\mathcal{L}(\mathcal{N})$ is the set of words $w \in \mathfrak{D}^\star$ such that either $w = \varepsilon$ or $w = a_0 \dots a_k$ and there exist a sequence of transitions $(q_i, \psi_i, q_{i+1}) \in \Delta$ and $a_i \in [\![\psi_i]\!]$ for $i \in [0, k]$, with $q_0 \in I$ and $q_k \in F$.

Example 8. *Consider again the Boolean algebra of Example 7. Figure 25 illustrates three examples of s-ϵNFAs, $\mathcal{M}_{pos}$, $\mathcal{M}_{ev/odd}$, and $\mathcal{M}_{ev/odd} \times \mathcal{M}_{pos}$, built on this algebra and where $\psi_{>0}(x) \wedge \psi_{odd}(x)$ denote the formula from Example 7. The automaton $\mathcal{M}_{pos}$ accepts all strings consisting only of positive numbers, while $\mathcal{M}_{ev/odd}$ accepts all strings of even length consisting only of odd numbers. For example, $\mathcal{M}_{ev/odd}$ accepts the string $2, 4, 6, 2$ (we separate characters with a comma for clarity, but the comma is not in the language) and rejects the strings $2, 4, 6$ and $51, 26$. The automaton $\mathcal{M}_{ev/odd} \times \mathcal{M}_{pos}$ is obtained by the product of $\mathcal{M}_{pos}$ and $\mathcal{M}_{ev/odd}$, and accepts the language $\mathcal{L}(\mathcal{M}_{pos}) \cap \mathcal{L}(\mathcal{M}_{ev/odd})$.*

Tamm and Veanes [81] proved that we can apply the counterpart of Thompson's construction [82, 83] to a symbolic regular expression to derive an *s-ϵNFA*. For further details on symbolic regular expressions and automata, we refer the interested reader to [81].

Now we define $\mathcal{P} = (\mathfrak{D}, \Psi, [\![\_]\!], \top, \bot, \vee, \wedge, \neg)$ the Boolean algebra for our policies, as follows:

- $\mathfrak{D}$ is the set of ground patterns generated by the first production *patt* of Figure 19, namely, $\mathfrak{D} = \{p \in \mathit{patt} \mid p = \mathtt{C}.\mathtt{f}(\boldsymbol{v})\}$.
- Ψ is the set of pattern predicates defined as follows. For each pattern $p \in \mathit{patt}$ we introduce a predicate $\psi_p(x) \triangleq x = p$ that holds when the argument x (a ground pattern from $\mathfrak{D}$) matches p for

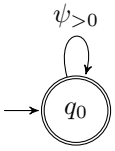

Figure 22: Automaton $\mathcal{M}_{pos}$.

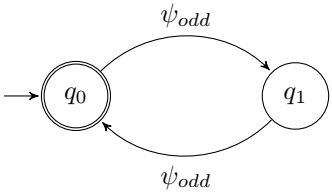

Figure 23: Automaton $\mathcal{M}_{ev/odd}$.

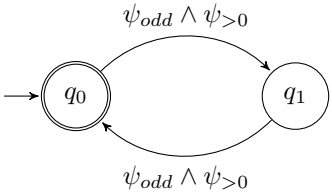

Figure 24: Automaton $\mathcal{M}_{ev/odd} \times \mathcal{M}_{pos}$.

Figure 25: Symbolic automata built on the Boolean algebra of Example 7.

some substitution θ of variables in p. Therefore, we define the set of predicates $\Psi = \{\psi_p \mid \psi_p \text{ for } p \in patt\}$ and we close it under logical connectives $\vee, \wedge, \neg$.

- The meaning of a pattern predicate ψ_p is $[\![\psi_p]\!] = \{q \mid \psi_p(q) \text{ holds for some } \theta\}$.

We note that the Boolean algebra $\mathcal{P}$ is effective because $[\![\psi_p]\!] \neq \emptyset$ is decidable. Indeed, at least two approaches exist to check the satisfiability of a pattern predicate ψ_p. The first approach involves a straightforward matching between the argument x (a ground pattern) and the pattern p, finding a possible assignment to the variables of p that makes them syntactically equal. The second approach instead involves encoding our Boolean algebra in the theory of the uninterpreted function with equality and using a standard SMT solver to check satisfiability.

To ensure that a history η complies with a policy $\phi = (re, E)$ within the contract $\mathfrak{X}$ and in the state σ', in symbols $\eta' \vDash_{\sigma',\rho,\mathfrak{X}} \phi$, we proceed as follows. First, we transform re into a symbolic regular expression predicating on the pattern of the policy and build the corresponding *s-ϵNFA* $\mathcal{N}$. Proposition 1 of [81] ensures that the *s-ϵNFA* accepts the same language of the symbolic regular expression. Then we check if the history η' belongs to the language $\mathcal{L}(\mathcal{N})$: this means that if there exists a sequence of transitions $q_0 \xrightarrow{p_1} \dots \xrightarrow{p_n} q_f$ for $\eta' = p_1 \dots p_n$ where each transition is possible for some substitution θ_i. Therefore, η' belongs to the language $\mathcal{L}(\mathcal{N})$ holds for some $\theta' = \theta_1 \dots \theta_n$. Then, we evaluate the assertion $[\![E]\!]_{\sigma',\rho\cdot\theta'}$ in the state σ' and in the environment ρ extended with the substitution θ' and check that it evaluates to true. If all the steps above succeed, η' complies with ϕ, otherwise, it does not: when the history η' does not belong to the language of $\mathcal{N}$ ($\eta' \notin \mathcal{L}(\mathcal{N})$) or E evaluates to false, the execution is aborted because of a policy violation.

Example 9. *Consider a policy $\phi = (re, E)$ where $re = x.\mathtt{f}_1(y) \cdot z.\mathtt{f}_2(w)^\star$ and $E = (y > 0 \wedge w = 13)$. We first transform re into a symbolic regular expression sre as follows: take the predicates $\psi_1 = x.\mathtt{f}_1(y)$ and $\psi_2 = z.\mathtt{f}_2(w)$, then $sre = \psi_1 \cdot \psi_2^\star$ by simply lifting the patterns of re in the corresponding predicates. From this symbolic regular expression, we build the corresponding automaton $\mathcal{N}$ shown in Figure 26. Given a history $\eta = c.\mathtt{f}_1(18) \cdot d.\mathtt{f}_2(13) \cdot d.\mathtt{f}_2(13)$, it is easy to check that $\eta \in \mathcal{L}(\mathcal{N})$ with variables assignment $\theta = \{y \mapsto 18, x \mapsto$*

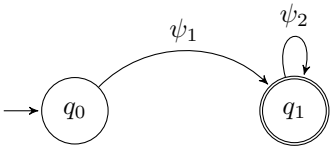

Figure 26: Automaton $\mathcal{N}$ for the symbolic regular expression sre of Example 9.

$c, z \mapsto d, w \mapsto 13\}$, *so* η *satisfies* re. *Then, we evaluate* E *in a state* σ *and environment* ρ *extended with* θ, *namely* $[\![E]\!]_{\sigma,\rho\cdot\theta} = (y > 0 \wedge w = 13)$ *that evaluates to true. Therefore, history* η *is compliant with the policy* ϕ.

5.3 Use Cases

In this section, we provide some relevant use cases that show how our policy framework works. Our mechanism detects vulnerabilities at the level of transaction execution or smart contract interaction. These attacks can be difficult to detect as they often exploit the intended functionality of smart contracts rather than explicit coding errors.

5.3.1 Use Case 1: Detecting Reentrancy

Here, we demonstrate how our policy framework can detect and prevent reentrancy attacks, specifically, mono-function and cross-chain reentrancy, using a history-based enforcement mechanism.

Example 1: Detecting Mono-function Reentrancy

Figure 18 illustrates a mono-function reentrancy scenario. In Section 5.1, we introduce a policy to protect the call to the `withdraw()` function, shown in Listing 5.1. The objective is to capture recursive invocations of this function within the same transaction context and block execution when reentrancy conditions are met.

The vulnerability arises because the transfer to `msg.sender` occurs before the user's balance is updated. If `msg.sender` is a contract with

a fallback function, it may re-enter `withdraw()` before the state update takes place, allowing multiple withdrawals of the same balance.

The following sequence of calls exemplifies a mono-function reentrancy attack:

1. `Alice.deposit(1ETH)`
2. `Bob.deposit(1ETH)`
3. `Kate.deposit(1ETH)`
4. `Mallory.attack()`
5. `ChainReentrancy.deposit(1ETH)`
6. `ChainWallet.withdraw()`
7. `ChainWallet.sendFunds(1ETH)`
8. `ChainReentrancy.receive()`
9. `ChainWallet.withdraw()`
10. Repeat 6–8 to perform multiple reentrant calls
11. `ChainWallet.updateBalance()`

Mono-function reentrancy is a powerful exploit that can be mitigated using runtime policy enforcement. By monitoring the sequence of function invocations, our framework prevents the contract from re-entering its own vulnerable function before completing state updates.

In detail, to prevent this behavior, we wrap the vulnerable `withdraw()` function in a policy frame, as shown in Listing 5.2. The policy dynamically monitors the execution history and blocks reentrant calls by detecting repeated invocations from the same address within the same call context.

Listing 5.2: Policy introduction in the `withdraw()`.

```
function withdraw() public {
    (φ) {
        uint amount = balances[msg.sender];
        if (amount > 0) {
            (bool success, ) = msg.sender.call{value: amount}("");
            require(success, "Transfer failed.");
            balances[msg.sender] = 0;
        }
    }
}
```

The regular expression re matches a trace consisting of two `withdraw()` invocations on the same callee contract (c), issued by two callers ($x \neq x'$), with the two withdrawals interleaved by `sendFunds()` followed by `receive()`. The complete sequence of calls of a mono-reentrancy attack vector is the following:

- `c.withdraw()`
- `c.sendFunds()`
- `d.receive()`
- `c.withdraw()`

The assertion E verifies that the `withdraw()` calls are made by two callers ($x \neq x'$).

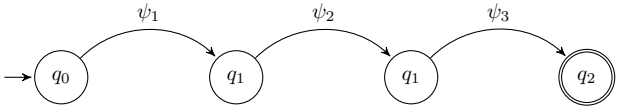

Figure 27: Automaton $\mathcal{A}_{monoR}$ preventing mono-function reentrancy behavior.

Formally, the policy $\phi = (re, E)$ is defined as follows:

$$re \;=\; c.\texttt{withdraw}(x) \;\cdot\; \alpha^* \;\cdot\; c.\texttt{withdraw}(x')$$
$$E \;=\; x \neq x'$$

The regular expression re captures two invocations of `withdraw()` on the same contract c by addresses x and x' interleaved with α^*, an arbitrary finite (possibly empty) sequence of actions. Here, it denotes $\alpha^* = c.\texttt{sendFunds}(x) \cdot x.\texttt{receive}()$. The assertion E ensures that both calls are issued by different addresses ($x \neq x'$). This reflects a prevention of a typical reentrant pattern, where the function is entered recursively before completion.

We transform re into the symbolic regular expression $\psi_1 \cdot \psi_2 \cdot \psi_3$, where:

$$\psi_1 = c.\texttt{withdraw}(x), \quad \psi_2 = c.\texttt{sendFunds}(x) \cdot x.\texttt{receive}(c)$$
$$\psi_3 = c.\texttt{withdraw}(x')$$

In our example, we have this execution history:

$$\begin{aligned}\eta = &\underline{\texttt{ChainWallet.withdraw}(\texttt{ChainReentrancy})}\cdot\\ &(\texttt{ChainWallet.sendFunds}(\texttt{ChainReentrancy})\cdot\\ &\texttt{ChainReentrancy.receive}(\texttt{ChainWallet}))\cdot\\ &\underline{\texttt{ChainWallet.withdraw}(\texttt{ChainReentrancy})}\end{aligned}$$

We observe that $\eta \in \mathcal{L}(\mathcal{A}_{monoR})$ with the variable assignment:

$$\theta = \{c \mapsto \texttt{ChainWallet},\ x \mapsto \texttt{ChainReentrancy},\ x' \mapsto \texttt{ChainReentrancy}\}$$

In this context, c denotes the vulnerable contract `ChainWallet`, while x is the address of the caller contract `ChainReentrancy`.

The term `callDepth`(x) refers to the current call stack depth associated with the execution context of x, and the operator $\cdot$ denotes sequential composition in the execution trace.

We then evaluate the assertion E in a state σ and environment ρ extended with θ, i.e., $[\![E]\!]_{\sigma,\rho\cdot\theta}$, that evaluates to false because the caller addresses are the same ($x = x'$) and the second call occurs before the state update so the call depth is greater than 1.

In our example, the attacker re-enters the `withdraw()` function via a fallback call before the original invocation completes. Since the internal state update occurs only at the end of the function, the contract still perceives the attacker's balance as unchanged during the reentrant call. This behavior violates the policy condition designed to block such recursive access patterns.

Referring to the sequence of calls in Listing 5.3.1, we observe that steps 6 and 9 correspond to two consecutive invocations of `withdraw()`, separated by intermediate calls to `sendFunds()` and `receive()`. The

attacker effectively re-enters the vulnerable function before the state is correctly updated, exploiting the contract's failure to finalize internal changes before making external calls.

This pattern precisely confirms the attempt of a reentrancy trace specified by the regular expression re, and thus the execution history is rejected. As a result, the history η is not compliant with the policy ϕ, and the execution call of the first `withdraw()` is reverted, thereby successfully preventing the exploit. Using our runtime policy enforcement mechanism, it is also possible to detect other forms of reentrancy, such as cross-function, create-based, and delegate-based reentrancy, by applying the same approach we used for mono-function reentrancy.

Example 2: Detecting Cross-Chain Reentrancy

Cross-chain reentrancy is an emerging class of reentrancy attacks, gaining attention with the increasing adoption of cross-chain messaging protocols. The growing shift toward interoperability and a multi-chain future necessitates careful consideration of this threat model. Protocols that bridge assets or rely on cross-chain communication must account for interactions across independent contract instances deployed on different chains.

Unlike traditional single-chain security models, the cross-chain setting introduces new challenges, as assumptions that hold within a single context may no longer apply. For this reason, mechanisms considered safe on one chain may lead to vulnerabilities when replicated across chains, as demonstrated by replay attacks.

Here we illustrate a cross-chain reentrancy attack vector and show how our history-based policy framework can detect and prevent such exploits. Cross-chain reentrancy arises when a contract issues an unsafe external call (such as the ERC-721 receiver hook `onERC721Received`) before committing its own state, allowing the callee to re-enter and initiate a cross-chain transfer path that later completes on another chain via bridge validators. ERC-721 [84] is the Ethereum standard for non-fungible tokens (NFTs), defining interfaces for uniquely identifiable, non-divisible assets, including ownership, safe transfer, and metadata.

We introduce a representative NFT bridging architecture between `ChainA` and `ChainB`. This design enables the deployment of the same NFT contract on both chains and treats them as a single logical collection, with a bridge forwarding authenticated cross-chain messages. This guarantees that each token ID has exactly one owner at any time and prevents duplication (which would otherwise make the asset effectively fungible). To enable mobility, the contract provides a dedicated cross-chain transfer primitive.

In our example, the NFT collection is `crossChainWarriors`. As shown in Figure 28, moving a warrior from `ChainA` to `ChainB` follows a burn–mint workflow: the token is burned on `ChainA` and, under bridge control, re-minted on `ChainB`.

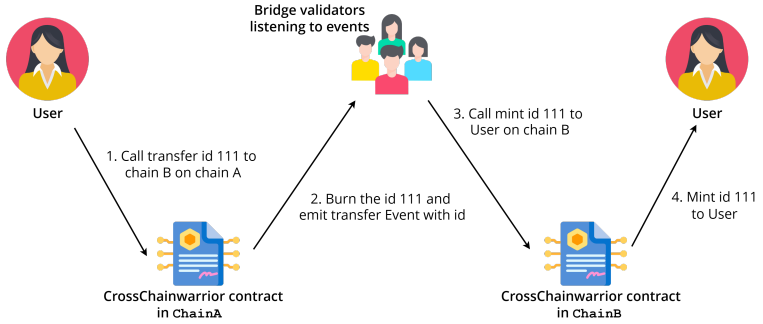

Figure 28: Bridge abstraction and burn–mint transfer from `ChainA` to `ChainB`.

Calling `crossChainTransfer()` on `ChainA` emits an event that bridge validators observe, attest, and relay to the destination chain; they then invoke the destination contract with the verified payload.

On `ChainB`, the `mint()` in Listing 5.3 calls `_safeMint()` to issue the NFT.

Listing 5.3: `crossChainWarriors mint` function.

```
function mint(address to) public returns (uint256) {
    uint256 newWarriorId = tokenIds.current();
    _safeMint(to, newWarriorId);
    tokenIds.increment();
    return newWarriorId;
```

```
}
```

Note that `_safeMint()` in Listing 5.4 has the same logic as the `mint` function and performs an extra check to ensure that the recipient can handle ERC721 functionality, meaning user-controlled code may execute during the callback, and the ordering is security-relevant.

Listing 5.4: `_safeMint` function.

```
function _safeMint(
    address to,
    uint256 tokenId,
    bytes memory data
) internal virtual {
    _mint(to, tokenId);
    require(
        _checkOnERC721Received(address(0), to, tokenId, data),
        "ERC721: transfer to non ERC721Receiver implementer"
    );
}
```

`_safeMint` function provides an extra check to secure users from losing their tokens, which only verifies that the receiver is aware of `IERC721Receiver` and does not guarantee that the recipient can handle ERC-721.

Listing 5.5: `_checkOnERC721Received` function.

```
1  function _checkOnERC721Received(
2      address from,
3      address to,
4      uint256 tokenId,
5      bytes memory data
6  ) private returns (bool) {
7      if (to.isContract()) {
8          try IERC721Receiver(to).onERC721Received(
9              _msgSender(), from, tokenId, data
10         ) returns (bytes4 retval) {
11             return retval == IERC721Receiver.onERC721Received.selector;
12         } catch (bytes memory reason) {
13             if (reason.length == 0) {
14                 revert("ERC721: transfer to non ERC721Receiver
                       implementer");
15             } else {
16                 /// @solidity memory-safe-assembly
17                 assembly {
18                     revert(add(32, reason), mload(reason))
19                 }
20             }
21         }
22     } else {
23         return true;
```

```
24         }
25     }
```

In Listing 5.5, line 8 issues the external call that probes whether the recipient implements `IERC721Receiver`. To pass the check, the callee must return `IERC721Receiver.onERC721Received.selector`; however, the hook may execute arbitrary logic before returning (including re-entering the caller), so the call ordering is security-relevant. Listing 5.6 shows an example of a possible contract that correctly implements the ERC-721 receiver interface.

Listing 5.6: A minimal ERC-721 receiver.

```
contract SimpleReceiver is IERC721Receiver {
    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external pure returns (bytes4) {
        // Optional: custom behavior
        return this.onERC721Received.selector;
    }
}
```

If the recipient is the malicious contract in Listing 5.7, its `onERC721Received()` hook can re-enter the transfer flow and trigger a second `mint()` before the prior state update completes.

Listing 5.7: Cross-chain reentrancy attacker.

```
1  contract Attacker is IERC721Receiver {
2      bool public _hasReentered;                    // prevent infinite
           loop
3      address _beneficiary;
4      CrossChainWarriors internal _contractChainA; // origin (Chain A)
5      CrossChainWarriors internal _contractChainB; // destination (Chain
           B)
6
7      constructor(address contractChainA, address contractChainB) {
8          _beneficiary = msg.sender;
9          _contractChainA = CrossChainWarriors(contractChainA);
10         _contractChainB = CrossChainWarriors(contractChainB);
11     }
12
13     function onERC721Received(
14         address,
15         address,
16         uint256 tokenId,
17         bytes calldata
18     ) external override returns (bytes4) {
```

```
19          if (!_hasReentered) {
20              // Re-enter cross-chain flow, then mint again
21              _contractChainA.crossChainTransfer(2, _beneficiary, tokenId
                    );
22              _hasReentered = true;
23              _contractChainA.mint(_beneficiary);
24          }
25          return this.onERC721Received.selector;
26      }
27  }
```

This can duplicate the same `tokenId`, violating NFT uniqueness and undermining the bridge's safety guarantees.

The following sequence of calls exemplifies a cross-chain reentrancy attack:

1. `ChainA.NFT.crossChainTransfer(User, tokenId)`
2. `Bridge.relay(ChainA, ChainB, tokenId)`
3. `ChainB.NFT._safeMint(to, tokenId)`
4. `Recipient.onERC721Received(ChainB.NFT, tokenId)`
5. `ChainB.NFT.crossChainTransfer(tokenId)`
6. `ChainB.NFT._safeMint(to, tokenId)`
7. `ChainB.NFT.increment();`

In the call sequence, the vulnerable `mint()` on `ChainB` should be wrapped by a history-based enforcement policy to block recursive cross-chain minting. In Step 6, the (`ChainB.NFT._safeMint(to, tokenId)`) function is the reentrant entry point and the primary site for enforcement.

Listing 5.8: Policy-protected `mint()` on `ChainB`.

```
function mint(address to) public returns (uint256) {
  (φ) {
       uint256 newWarriorId = tokenIds.current();
      _safeMint(to, newWarriorId);
      tokenIds.increment();
      return newWarriorId;
    }
  }
```

Formally, the policy $\phi = (re, E)$ is defined as follows:

$$re \;=\; c.\texttt{mint}(x) \;\cdot\; \texttt{call}(x) \;\cdot\; x.\texttt{onERC721Received}() \;\cdot\; c.\texttt{mint}(x'),$$
$$E \;=\; x \neq x'.$$

The assertion E ensures that both calls are issued by different addresses ($x \neq x'$). This reflects a prevention of a reentrant pattern, where the function is entered recursively before completion. We transform re into the symbolic regular expression $\psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \psi_4$, where:

$$\psi_1 = c.\texttt{mint}(x), \quad \psi_2 = b.\texttt{call}(x),$$
$$\psi_3 = x.\texttt{onERC721Received}(), \quad \psi_4 = c.\texttt{mint}(x')$$

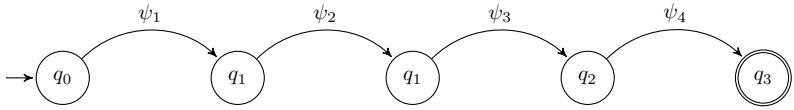

Figure 29: Automaton $\mathcal{A}_{crossR}$ preventing cross-chain reentrancy pattern.

Given the execution history:

$$\eta = \texttt{ChainB.mint}(\texttt{Attacker}) \cdot \texttt{ChainB.call}(\texttt{Attacker}) \cdot$$
$$\texttt{Attacker.onERC721Received}() \cdot \texttt{ChainB.mint}(\texttt{Attacker})$$

we observe that $\eta \in \mathcal{L}(\mathcal{A}_{crossR})$ with the variable assignment:

$$\theta = \{c \mapsto \texttt{ChainB},\ b \mapsto \texttt{Bridge},\ x \mapsto \texttt{Attacker},\ x' \mapsto \texttt{Attacker}\}$$

We then evaluate the assertion E in a state σ and environment ρ extended with θ, namely $[\![E]\!]_{\sigma,\rho\cdot\theta}$, which evaluates to false because the reentrant caller coincides with the original one ($x = x'$).

This pattern confirms the attempt of a reentrancy trace specified by the regular expression re, and thus the execution history is rejected. As a result, the history η is not compliant with the policy ϕ, and the execution call of the first `mint()` is reverted, thereby successfully preventing the exploit.

Our policy framework detects the cross-chain reentrancy by tracing a reentrant call sequence across contracts and enforcing call depth constraints. Even in distributed or asynchronous bridging protocols, our approach ensures correctness by halting recursive execution that violates the defined invariants.

5.3.2 Use Case 2: Detecting Price Manipulation

We now show our policy framework can detect both *direct* and *indirect* price manipulation attacks in DeFi protocols.

A *direct price manipulation* (DPM) attack occurs when an attacker explicitly alters the price of a token within a liquidity pool of an AMM often to gain an unfair advantage or extract profit.

The mechanism of a DPM attack typically involves corrupting or influencing the oracle's reported price. Since oracles act as external sources of crypto-asset valuations, attackers may inject false or misleading data into them, causing the smart contract to believe in a manipulated market state. Once the oracle-reported price deviates from its true value, the attacker can exploit this discrepancy by executing trades at artificially advantageous rates.

For instance, an attacker may buy or sell large volumes of tokens, temporarily inflating or deflating their price, and then leverage the manipulated rate for arbitrage or liquidation purposes. In some cases, the attacker interacts directly with the target protocol's pool to distort its internal pricing curve - using, for instance, a flash loan, enabling profitable outcomes through the induced slippage or mispriced exchange rate.

In contrast, an *indirect price manipulation* (IPM) attack affects the target protocol without directly interacting with it. Instead, the attacker manipulates an external pricing source that is typically a separate AMM or oracle, on which the victim protocol relies. For instance, if a lending platform calculates collateral ratios using the spot price from an external AMM an attacker can alter that price via an off-platform trade, thereby indirectly influencing the behavior of the victim contract. The goal is to artificially inflate the value of the attacker's collateral. This allows them to mint more LP tokens as proof of deposit, enabling excessive borrowing of liquid assets. Since the victim protocol treats the manipulated price as valid, the attacker can drain funds and exit the system, leaving it undercollateralized and causing losses to users and the protocol itself.

In both DPM and IPM settings, our history-based policy can identify attack-characteristic call traces and applies runtime constraints that

halt execution when such patterns arise. Enforcing these policies enables DeFi protocols to preempt composability-driven exploits that would otherwise evade static analyses and simple slippage guards.

Example 3: Detecting Arbitrage

Flash loans offer several financial opportunities for DPM attacks and speculative actions, including arbitrage. Milionis *et al.* [85] provide an in-depth analysis of arbitrage focusing on the differences between AMMs and market prices. Based on their strategies, arbitrageurs can be classified into opportunistic and myopic. The first category waits for larger mispricings, hoping for greater future gains rather than immediate profits, and prefers missing opportunities for faster, more short-term focused arbitrageurs. The second one, on the other hand, aims to maximize immediate profit: users execute trades that make a profit by buying (or selling) crypto-assets from an AMM if their price, adjusted for fees, is below (or above) the market one, and then selling (or buying) them to another AMM at the market price. Listing 5.9 shows prototypical flash loan pseudocode in the style of Aave [15]: the pool lends funds, the borrower executes arbitrary logic via an external callback, and the loan *must* be repaid (plus fee) before the transaction ends - otherwise the whole transaction reverts (see Chapter 2 for an in-depth introduction to flash loans).

Listing 5.9: Pseudocode for a `flashLoan` function illustrating liquidity provision and repayment logic.

```
function flashLoan(receiverAddress, asset, amount, params) {
    // Calculate fees for the asset
    // Check if there is enough liquidity
    if (liquidity[asset] < amount) {
        revert("Insufficient liquidity");
    }

    // Update the liquidity to reflect the loan
    liquidity[asset] -= amount;

    // Transfer the assets to the receiver
    transfer(asset, receiverAddress, amount);

    // Execute the operation in the receiver contract
    bool success = receiverAddress.executeOperation(asset, amount, fee,
        msg.sender, params);
```

```
16      if (!success) {
17          revert("FlashLoan execution failed");
18      }
19
20      // Collect the borrowed amount plus fee
21      uint256 totalDebt = amount + fee;
22
23      // Check if receiver has enough tokens to repay the loan
24      uint256 receiverBalance = getBalance(receiverAddress, asset);
25      if (receiverBalance < totalDebt) {
26          revert("Insufficient token balance to repay the loan");
27      }
28
29      transferFrom(receiverAddress, self, totalDebt, asset);
30      liquidity[asset] += totalDebt;
31
32      // Emit the flash loan event
33      emit FlashLoan(receiverAddress, asset, amount, fee);
34  }
```

The receiver contract borrows, runs `executeOperation`, and must return the principal plus fees within the same transaction; otherwise, the loan is rolled back. Parameters are: `asset` (token address), `amount` (borrowed quantity), and `params` (auxiliary data for the strategy).

A typical myopic flash loan arbitrage proceeds as follows: borrow on one venue, perform a sequence of swaps across other AMMs to exploit price discrepancies, and repay - capturing the spread.

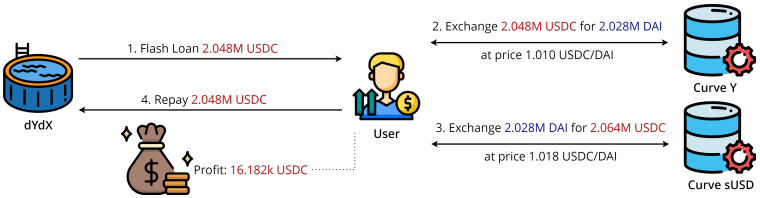

Figure 30: High-level execution of a flash loan-based arbitrage transaction: (1) flash loan requested to the dYdX liquidity pool; (2) swap exchange from USDC to DAI in the AMM Curve Y; (3) swap exchange from DAI to USDC in the AMM Curve sUSD;(4) repay the flash loan.

Figure 30 illustrates the steps of a flash loan-based arbitrage involving the liquidity pool dYdX and the two AMMs, CurveY and Curve sUSD. The transaction dated back to July 31, 2020,[2] the arbitrageur borrowed

[2]Its address is 0xf7498a2546c3d70f49d83a2a5476fd9dcb6518100b2a731294d0d7b9f79f754a.

2.048 million USDC,[3] executed two exchanges, one from USDC to DAI[4] from Curve Y, and the other one in the opposite directions from Curve sUSD. At the end of them, she paid its debt with dYdX but realized a profit of 16.182 thousand USDC.

Arbitrage, while beneficial for synchronizing asset prices across different DeFi markets, can be exploited to perform attacks on the market as described above. The capability of composing DeFi protocols through external calls is essential due to the interconnected nature of DeFi platforms. However, this composability not only facilitates chained trading and arbitrage opportunities but also introduces opportunities for malicious actors. Therefore, regulating external calls through robust policies is imperative to mitigate potential attacks and maintain market integrity. Here, we show how history-based policies can regulate how the various protocols are composed, enforcing good behavior among the involved parties.

Protecting the invocation of the function `executeOperation` with a policy ϕ in the function `flashLoan` enables us to identify and potentially mitigate profitable unwanted scenarios like the one described above.

Here, we show how our policy framework can prevent the arbitrage shown in Figure 30 and illustrates how our policy checking mechanism works. We introduce a policy to protect the call to `executeOperation()` in Listing 5.9 for detecting sequences of function calls that may indicate an attempt at arbitrage. The idea is that our policy detects sequences of invocations to the `swap` method to transfers of tokens in opposing directions involving the same token types and different AMMs. The policy will be parametric on the contracts performing the transfers and on the types of tokens. Below, we denote with `User`, `dYdX`, `CurveY`, and `CurvesUSD` the addresses of the contracts involved in Figure 30 and with `Flash` the address of the contract created by the user implementing the `executeOperation` function. Moreover, we use τ_a and τ_b to indicate

[3] USD Coin is a stablecoin entirely backed by U.S. dollar assets: each USDC token represents a tokenized U.S. dollar, with a value closely mirroring that of one U.S. dollar.

[4] DAI is an Ethereum stablecoin that maintains a value close to USD using supply-controlling smart contracts, regulated by MakerDAO's MKR token holders.

the token types USDC and DAI, respectively. Also, we assume that the market price for token types τ_a and τ_b is provided by an Oracle contract with address `0`. Assume `User` requests a `loan` of 2048 million tokens of type τ_a and invokes the `flashLoan` function of Listing 5.10 passing the address of `Flash` as first parameter:

Listing 5.10: Function implementing an arbitrage using flash loan swaps on CurveY and CurvesUSD.

```
function executeOperation(
    address asset,
    uint256 amount,
    uint256 fee,
    address user,
    bytes memory params
) external {
    // First Swap on CurveY
    CurveY.swap(user, USDC, DAI, 2048e6);

    // Get New Token Balance after the first swap
    uint256 toBalance = IERC20(DAI).balanceOf(address(this));
    require(toBalance > 0, "Swap on CurveY failed to return any tokens");

    // Second Swap on CurvesUSD
    CurvesUSD.swap(user, DAI, USDC, 2028e6);

    // Get New Token Balance after the second swap
    uint256 finalBalance = IERC20(USDC).balanceOf(address(this));
    require(finalBalance > 0, "Swap on CurvesUSD failed to return any
        tokens");

    // Ensure that final balance is enough to repay the loan with fee
    require(finalBalance >= amount + fee, "Arbitrage trade did not return
        enough tokens to repay the loan");
}
```

The execution of `flashLoan()` results in the following sequence of calls:

1. `dYdX.balance()`
2. `dYdX.transfer(`τ_a`, Flash, loan)`
3. `Flash.executeOperation()`
4. `CurveY.swap(User,` τ_a`,` τ_b`, amount`$_1$`)`
5. `CurvesUSD.swap(User,` τ_b`,` τ_a`, amount`$_2$`)`
6. `Flash.payback(dYdX)`

First, the `dYdX` contract performs some sanity checks, during these checks, it invokes the `dYdX.balance()` function. Then, the requested amounts of tokens are transferred to the contract `Flash` via a call to

dYdX.transfer(τ_a, Flash, loan). This transfer of tokens is followed by the invocation of executeOperation() of the Flash contract that executes the two swaps: the first swap concerns $\texttt{amount}_1$ (2048 million) of tokens of type τ_a to acquire $\texttt{amount}_2$ (2028 million) tokens of type τ_b on CurveY (CurveY.swap(User, τ_a, τ_b, $\texttt{amount}_1$)). Then, the function performs the reverse swap on CurvesUSD (CurvesUSD.swap(User, τ_b, τ_a, $\texttt{amount}_2$)). After these swaps, the Flash contract repays its debt to the dYdX, including additional fees, via the call to the function Flash.payback(dYdX).

Listing 5.11: Annotated Solidity snippet: checking receiver contract execution with ϕ.

```
1 // Execute the operation in the receiver contract
2 (φ) {
3     bool success = receiverAddress.executeOperation(asset, amount, fee,
          msg.sender, params);
4     if (!success) {
5         revert("FlashLoan execution failed");
6     }
7 }
```

We prevent the execution of the arbitrage by enclosing the call to the function executeOperation inside a policy framing construct. The amended line of code of the flashLoan function is shown in Listing 5.11. The policy ϕ blocks the execution when it detects a sequence of swaps of opposing directions involving the same token types across different AMMs (steps 4 and 5 of the flow described above), that AMMs have a different token ratio, and that the token price significantly differs from their market price provided by the oracle 0. Formally, the policy $\phi = (re, E)$ is defined as follows:

$$
\begin{aligned}
re =& a.\texttt{swap}(x, y, z) \cdot b.\texttt{swap}(x', z', y') \\
E =& (a \neq b \wedge x = x' \wedge \frac{a.y}{a.z} = \frac{b.z'}{b.y'} \wedge \left|\frac{a.y - \texttt{0}.y}{a.z - \texttt{0}.z}\right| \leq \epsilon \wedge \left|\frac{b.z' - \texttt{0}.z'}{b.y' - \texttt{0}.y'}\right| \leq \epsilon)
\end{aligned}
$$

where the regular expression re detects the sequence of the two consecutive swaps, while the assertion E holds when the involved AMMs are different ($a \neq b$), the calling contract is the same ($x = x'$), the ratio between the exchanged token is the same ($\frac{a.y}{a.z} = \frac{b.z'}{b.y'}$), and the ratio of the difference between the exchanged tokens of the AMMs and the market price of each asset as given by contract Oracle 0 to is below a small

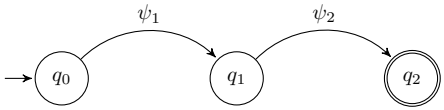

Figure 31: Automaton $\mathcal{A}_{DPM}$ for the policy preventing flash loan-based arbitrages.

threshold ϵ. Note that in the assertion E we assume that the amount of the token type y exchanged through a swap on the AMM a is accessed using the syntax $a.y$. We use the same syntax for the other token types and AMMs, and for accessing the market price of each token type via the oracle `O`. We first transform re into the symbolic regular expression $\psi_1 \cdot \psi_2$ where $\psi_1 = a.\texttt{swap}(x, y, z)$ and $\psi_2 = b.\texttt{swap}(x', z', y')$, and then build the corresponding automaton $\mathcal{A}_{DPM}$ shown in Figure 31. Given $\eta = \texttt{CurveY.swap}(\texttt{User}, \tau_\texttt{a}, \tau_\texttt{b}, \texttt{amount}_1) \cdot \texttt{CurvesUSD.swap}(\texttt{User}, \tau_\texttt{b}, \tau_\texttt{a}, \texttt{amount}_2)$, it is easy to check that $\eta \in \mathcal{L}(\mathcal{A}_{DPM})$ with variables assignment $\theta = \{a \mapsto \texttt{CurveY}, b \mapsto \texttt{CurvesUSD}, x \mapsto \texttt{User}, x' \mapsto \texttt{User}, y \mapsto \tau_\texttt{a}, z \mapsto \tau_\texttt{b}, y' \mapsto \tau_\texttt{b}, z' \mapsto \tau_\texttt{a}\}$. Then, we evaluate E in a state σ and environment ρ extended with θ, namely $[\![E]\!]_{\sigma,\rho\cdot\theta}$, that evaluates to false because the ratio of between the exchanged tokens ($\frac{\texttt{CurveY.}\tau_\texttt{a}}{\texttt{CurveY.}\tau_\texttt{b}} \neq \frac{\texttt{CurvesUSD.}\tau_\texttt{a}}{\texttt{CurvesUSD.}\tau_\texttt{b}}$) differs. Indeed, as shown in Figure 30, the price ratio for `CurveY` is 1.010 USDC/DAI, while for `CurvesUSD` is 1.918 USDC/DAI. This discrepancy confirms the attempt of an arbitrage, so the history η is not compliant with the policy ϕ, and the execution of `flashLoan` is reverted.

Example 4: Detecting Indirect Price Manipulation

In IPM attacks, attackers inflate the on-chain price of an asset used as collateral, allowing them to borrow more than the legitimate value of their holdings. These attacks typically target lending protocols that rely on real-time price feeds from AMMs, which are manipulable due to their reserve-based pricing. Unlike DPM attacks, this type of exploit generates profit not through asset reselling, but by exploiting a protocol's pricing mechanism to obtain undercollateralized loans. Figure 32 illustrates the

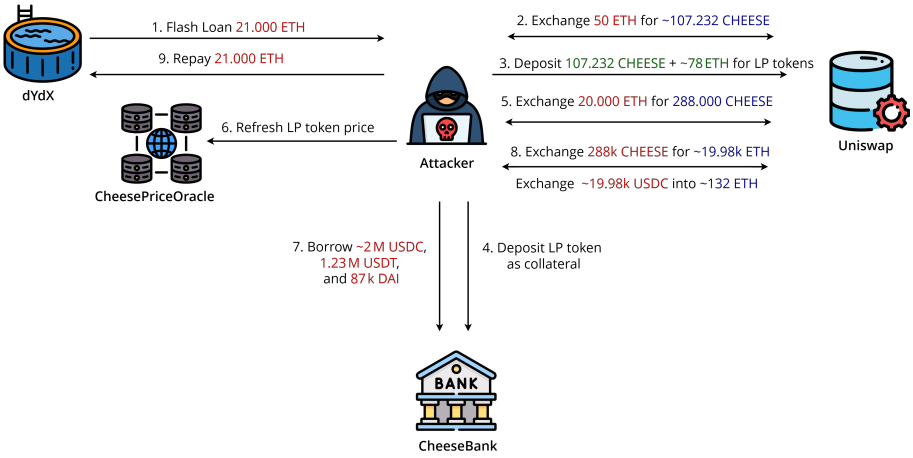

Figure 32: High-level execution of a flash loan-based IPM against Cheese Bank: (1) flash loan requested in ETH from a liquidity provider; (2) swap exchange from ETH to CHEESE in the AMM Uniswap; (3) add CHEESE and ETH liquidity in Uniswap to mint LP tokens; (4) submit LP tokens to Cheese Bank for collateral valuation; (5) swap exchange from ETH to CHEESE in Uniswap AMM to inflate LP valuation; (6) oracle/price read updates the LP price; (7) borrow stablecoins (USDC/USDT/DAI) from Cheese Bank against the inflated LP collateral; (8) unwind by swapping back and removing liquidity; (9) repay the flash loan.

Cheese Bank[5] incident [86, 87] of November 6, 2020, which resulted in losses exceeding $3.3M due to a flaw in the protocol's AMM-based oracle that priced LP collateral from instantaneous pool reserves. The core manipulation (Step 5) is a large ETH to CHEESE[6] swap on Uniswap that depletes the CHEESE reserve and raises the marginal CHEESE price in ETH, thereby inflating any LP–valuation function tied to the pool's ETH balance. The attacker then triggers an oracle/price read (Step 6), which ingests this distorted state and marks the LP tokens at the inflated level. Using the overvalued LP tokens as collateral, the attacker borrows sub-

[5]The Cheese Bank is a decentralized autonomous digital bank on Ethereum that provides services such as asset and fund management.

[6]CHEESE is a cryptocurrency available on DEXs (such as Uniswap) that is tailored to the needs of users, investors, and business owners. [88]

stantial stablecoin balances (e.g., USDC, USDT,[7] DAI), exceeding limits that would apply under unmanipulated prices (Step 7). Finally, the flash loan is repaid, leaving the pool close to its pre-attack reserves while the attacker retains the surplus withdrawn from Cheese Bank. The exploit succeeds because collateral valuation relied on manipulable spot AMM reserves without time-weighting or deviation bounds, and transaction atomicity allowed the distortion to be created, recognized, and monetized in a single execution.

The attack flow involves the following key sequence of actions:

1. `dYdX.flashLoan(ETH, fLoanAmount)`
2. `Flash.executeOperation()`
3. `Uniswap.swap(Attacker, ETH, CHEESE, seedAmount)`
4. `Uniswap.addLiquidity(Attacker, ETH, CHEESE, amounts)`
5. `CheeseBank.depositCollateral(Attacker, LPtokens)`
6. `Uniswap.swap(Attacker, ETH, CHEESE, manipulateAmount)`
7. `CheesePriceOracle.refresh()`
8. `CheeseBank.borrow(Attacker, {USDC, USDT, DAI}, bAmounts)`
9. `Uniswap.swap(Attacker, CHEESE, ETH, unwindAmount)`
10. `dYdX.repay(ETH, fLoanAmount*)` [8]

First, the `dYdX` contract creates a flash loan, transferring the requested amount of ETH to the attacker contract (`dYdX.flashLoan(ETH, fLoanAmount)`).[9] Upon receipt, `Flash` invokes `executeOperation()`, which coordinates the cross–protocol sequence. Inside `executeOperation()`, the contract acquires assets to supply liquidity by swapping a portion of ETH for CHEESE on Uniswap (`Uniswap.swap(Attacker, ETH, CHEESE, seedAmount)`), then adds ETH and CHEESE to the pool to mint LP tokens (`Uniswap.addLiquidity(Attacker, ETH, CHEESE, amounts)`). The initial vulnerability lies in Step 4, where there is collateral admission and valuation, and the freshly minted LP tokens are immediately eligible as collateral. Following this, the LP tokens are posted as collateral in Cheese Bank

[7] USDT (Tether) is a centralized stablecoin issued by Tether Limited that targets a 1:1 peg with the U.S. dollar, backed by reserve assets and widely used to move and trade value across multiple blockchains.

[8] The repayment of the flash loan includes the protocol fee.

[9] The flash loan mechanics and the associated call sequence are analyzed in the sub-Section 5.3.2.

(CheeseBank.depositCollateral(Attacker, LPtokens)), linking the position to the protocol's LP-valuation mechanism; if the valuation logic reads instantaneous AMM reserves, the posted LPs can be repriced within the same transaction. (Step 5). The main manipulation occurs in Step 6, where there is a large ETH to CHEESE swap on Uniswap to distort reserves and inflate the LP valuation linked to the pool's ETH balance (Uniswap.swap(Attacker, ETH, CHEESE, manipulateAmount). The attacker then triggers the price path, causing the distorted spot state to be ingested into Cheese Bank's accounting (CheesePriceOracle.refresh()), a vulnerability because the oracle reads deviation bounds and directly feeds valuation (Step 7). After this, the attacker drains stablecoins against the overvalued collateral (CheeseBank.borrow(Attacker, {USDC, USDT, DAI}, bAmounts)), and the protocol accepts inflated collateral or post–manipulation repricing (Step 8). Finally, the contract reverse the price impact by swapping CHEESE back to ETH (Uniswap.swap(Attacker, CHEESE, ETH, unwindAmount)) (Step 9) and pays the flash loan, returning the loan amount and fee to dYdX (dYdX.repay(ETH, fLoanAmount*)), ensuring the entire bundle either succeeds atomically or reverts (Step 10).

To mitigate such attacks, we propose a policy ϕ that monitors the execution history for suspicious sequences of liquidity manipulation and token valuation misuse. In particular, our policy detects cases where the price of LP tokens is artificially inflated by significant swaps prior to a sensitive call that depends on their valuation.

The critical external call is guarded by the following runtime policy:

Listing 5.12: Policy-enforced external call to LP token valuation logic.

```
// Evaluate LP token price-dependent external call under policy
(φ) {
    bool success = CheeseBank.externalCall(user, lpTokens);
    if (!success) {
        revert("External call rejected: suspicious LP token valuation");
    }
}
```

Formally, the policy $\phi = (re, E)$ is defined as:

$$
\begin{aligned}
re \;&= a.\texttt{swap}(x, y, z) \;\cdot\; a.\texttt{addLiquidity}(x, y', z') \;\cdot\; \mu \;\cdot\; a.\texttt{swap}(x, y'', z'') \cdot \\
&\quad\; \texttt{O}.\texttt{refresh}(x, y^*, z^*) \;\cdot\; b.\texttt{borrow}(x, w) \\
E \;&= (y = y' = y'' = y^*) \;\wedge\; (z = z' = z'' = z^*) \wedge \\
&\quad\; \left|\frac{a.y}{a.z} - \frac{\texttt{O}.y}{\texttt{O}.z}\right| \le \epsilon \;\wedge\; \left|\frac{a.y''}{a.z''} - \frac{\texttt{O}.y^*}{\texttt{O}.z^*}\right| \le \epsilon \;\wedge\; a.y' \le \iota \;\wedge\; a.z' \le \kappa\,.
\end{aligned}
$$

The regular expression re matches a `swap()` on the same AMM a, followed by `addLiquidity()` on a, then an arbitrary (possibly empty) subsequence μ of intermediate calls, a second `swap()` on a over the same token pair, an oracle `refresh()` on $\texttt{O}$, and finally a `borrow()` on b that consumes the LP-valued collateral w. The assertion E enforces token-pair coherence across all matched calls, price consistency of a with the oracle $\texttt{O}$ both before and after the liquidity addition (within tolerance ϵ), and small seeding amounts for the liquidity add, capped by ι and κ on $a.y'$ and $a.z'$, respectively.

The assertion E requires: *(i)* token-pair coherence between all the actions; *(ii)* price consistency of the AMM a with the reference oracle $\texttt{O}$ both immediately before and after the liquidity addition[10], meaning the AMM's implicit quote $a.y/a.z$ deviates from the oracle ratio $\texttt{O}.y/\texttt{O}.z$ by at most ϵ; and *(iii)* constrains the liquidity-seeding amounts by capping $a.y'$ and $a.z'$ at ι and κ, respectively.

We use the notation $a.t$ to denote the (post-call) amount or reserve of token t observed on AMM a, and $\texttt{O}.t$ to denote the oracle's reference price component for token t. Intuitively, if the execution trace satisfies re but violates E, for example when the AMM implied price diverges from the oracle by more than ϵ or when the liquidity-seeding amounts exceed ι and κ, the policy deems the trace manipulative and aborts the subsequent valuation-sensitive operation. We first transform re into the symbolic regular expression $\psi_1 \cdot \psi_2 \cdot \psi_3 \cdot \psi_4 \cdot \psi_5 \cdot \psi_6$ where $\psi_1 = a.\texttt{swap}(x, y, z)$, $\psi_2 = a.\texttt{addLiquidity}(x, y', z')$, $\psi_3 = \mu$, $\psi_4 = a.\texttt{swap}(x, y'', z'')$, $\psi_5 = \texttt{O}.\texttt{refresh}(x, y^*, z^*)$, and $\psi_6 = b.\texttt{borrow}(x, w)$.

[10]The deviation is evaluated after the call to `refresh()`, ($\texttt{O}.y^*/\texttt{O}.z^*$).

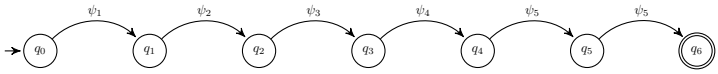

Figure 33: Automaton $\mathcal{A}_{IPM}$ for the policy preventing LP token price manipulation in Cheese Bank.

They are then transformed into the automaton $\mathcal{A}_{IPM}$ depicted in Figure 33, which accepts histories matching the manipulation pattern and enables runtime enforcement.

Given a transaction history:

$$\begin{aligned}\eta = {} & \texttt{Uniswap.swap}(\texttt{Attacker}, \texttt{ETH}, \texttt{CHEESE}, 50)\cdot \\ & \texttt{Uniswap.addLiquidity}(\texttt{Attacker}, (107.232\ \texttt{CHEESE} + 78\ \texttt{sETH}))\cdot\ \mu\cdot \\ & \texttt{Uniswap.swap}(\texttt{Attacker}, \texttt{ETH}, \texttt{CHEESE}, 20.000)\cdot \\ & \cdot \texttt{CheesePriceOracle.refresh}()\cdot \\ & \texttt{CheeseBank.borrow}(\texttt{Attacker}, \{2\text{m}\ \texttt{USDC}\ ,\ 1.23\text{m}\ \texttt{USDT}\ ,\ 87\text{k}\ \texttt{DAI}\})\end{aligned}$$

We verify that $\eta \in \mathcal{L}(\mathcal{A}_{\text{IPM}})$ under the variable assignment

$$\begin{aligned}\theta = \{\ & a \mapsto \texttt{Uniswap},\ b \mapsto \texttt{CheeseBank},\ x \mapsto \texttt{Attacker}, \\ & \texttt{0} \mapsto \texttt{CheesePriceOracle},\ (y, y', y'', y^*) \mapsto \texttt{ETH},\ (z, z', z'', z^*) \mapsto \texttt{CHEESE}\ \}\end{aligned}$$

The attacker raises the CHEESE price on Uniswap by swapping $20\,\text{k}$ ETH for about $288\,\text{k}$ CHEESE, which inflates the value of Uniswap LP tokens used as collateral at CheeseBank, since the protocol estimates the LP price from the pool's ETH balance. This manipulation enables draining via legitimate borrowing calls.

Then, we evaluate E in a state σ and environment ρ extended with θ, namely $[\![E]\!]_{\sigma,\rho\cdot\theta}$, that evaluates to *false* because the AMM implied price departs from the oracle by more than ϵ, i.e., $\left|\frac{a.y''}{a.z''} - \frac{\texttt{0}.y^*}{\texttt{0}.z^*}\right| > \epsilon$. Indeed, as shown in Figure 32, the Uniswap quote after the large ETH to CHEESE swap diverges materially from the oracle reference, reflecting reserve distortion. This discrepancy confirms an indirect price manipulation attempt, so the history η is not compliant with policy ϕ, and the execution of the guarded valuation call (e.g., `CheeseBank.borrow()`) is reverted.

5.4 Related Work

We compare our work with the relevant literature. Over the past few decades, the formalism of symbolic automata has been extensively studied [81, 89, 90], and applied to solve different verification tasks. Here, we rely on the results of Tamm and Veanes in [81] concerning the theory of symbolic regular languages and on the relation between symbolic automata and symbolic regular expression. To the best of our knowledge, this is the first paper that uses symbolic automata to express and enforce security policy following Schneider [73]. Automata-based techniques have also been successfully applied in other security domains: for instance, Black Ostrich [91] employs symbolic and alternating automata within SMT-based string solvers to reason about complex regular expressions arising in web application input validation. While this work demonstrates the effectiveness of automata-theoretic reasoning under symbolic constraints, it targets a fundamentally different application domain and threat model—namely web crawling and vulnerability discovery—and is therefore cited here solely as methodological inspiration rather than as directly related work on smart contract security. To formally present our policy mechanism, we extend TinySol, the minimal calculus for the Solidity contract introduced by Bartoletti *et al.* [75]. We enhanced the language by introducing a way to express policy over the execution history and the construct of policy framing to enforce them. Moreover, we extended the semantics accordingly and defined a procedure for checking policies at run-time.

The concept of history-based policies has been explored in previous works, notably by Skalka and Smith [92] and Bartoletti *et al.* [93, 94]. Skalka and Smith [92] combine type and effect systems with model-checking to extract a history expression from the code automatically and to verify that a higher-order program satisfies a policy ϕ. Their approach allows the static enforcement of history- and stack-based security mechanisms. Bartoletti *et al.* [93] uses a similar mechanism to express access control policies over resources and to verify that objects and resources are created and manipulated according to such policies. The main dif-

ference between these papers and ours is that their verification procedure occurs statically and on the code of programs, whereas our verification occurs at run-time. In particular, their verification procedures require the code of the whole program or at least an abstract description of the behavior of all called functions, given in terms of types and effects. Here, we do not have this assumption, so we can easily deal with open-world scenarios like DeFi, where a contract can be called by other unknown contracts. Several works have investigated the formal semantics and security analysis of Ethereum smart contracts. Grishchenko *et al.* [70, 95, 96], introduced a small-step operational semantics of EVM bytecode and used it to formally define fundamental security properties such as call integrity, atomicity, and single-entrancy. This semantic foundation was subsequently leveraged to design sound static analysis tools, most notably EtherTrust and eThor, which employ Horn-clause–based reachability analysis to automatically verify predefined security properties of EVM bytecode with formal soundness guarantees. These approaches focus on static verification and aim to prove the absence of certain classes of vulnerabilities. In contrast, our work addresses the complementary problem of regulating contract behavior at run time. Rather than verifying compliance with a fixed set of predefined properties, we provide a framework that allows developers to specify and enforce custom behavioral policies governing interactions with external contracts. This distinction is especially relevant in highly dynamic DeFi scenarios, where adversarial behaviors may depend on runtime values and execution contexts that are difficult to characterize statically. A large body of work addresses price manipulation and other vulnerabilities in DeFi and propose detection tools for smart contracts, such as DeFiRanger [97], DeFiScanner [98], DeFiTainter [99], BLOCKEYE [97], and FlashSyn [100]. DeFiRanger [97] builds a Cash Flow Tree (CFT) from Ethereum transaction data to reflect token transfers and accounts, giving a DeFi meaning to blockchain transactions. Wang *et al.* [98] uses taint analysis to uncover price manipulation vulnerabilities in DeFi protocols. BLOCKEYE [101] performs automated and accurate real-time attack detection of Ethereum-based DeFi protocols by symbolic reasoning tech-

niques. FlashSyn [100] automatically synthesizes adversarial transactions exploiting DeFi protocols via flash loans, overcoming complex DeFi logic and search space challenges with a synthesis-via-approximation technique. Unlike the above papers, our work focuses on formal verification, ensuring run-time security by validating histories produced by smart contract execution. This methodology guarantees the absence of security violations during execution, setting it apart from existing solutions. Another relevant runtime-verification framework is MoE [102], which models Ethereum attacks in Metric First-Order Temporal Logic and monitors transactions in real time (including near-miss variants) to detect evolving attack patterns. By contrast, our runtime policy reinforcement embeds lightweight, history-based automata directly around sensitive calls, actively reverting non-compliant executions on-chain rather than reporting them.

Finally, Zhou *et al.* [103] introduced a method based on analyzing internal transactions to detect and prevent attacks within DeFi platforms. They emphasize policy enforcement and vulnerability management amidst DeFi's composability, especially concerning arbitrage scenarios utilizing external calls. Although the goal of this paper is similar to ours, their focus is primarily on automated revenue extraction through trading strategies. In contrast, our smart contract calculus emphasizes formal methods and run-time verification for robust policy enforcement and vulnerability management.

Chapter 6

Conclusion

This thesis presents methods and insights that introduce formal models and mechanisms supporting liquidity-improvement approaches. Below, we briefly summarise the key contributions, findings, and implications, and outline directions for future work.

Chapter 3 introduced a liquidity-optimized mechanism that supports the typical interactions between users and AMMs, such as deposit, redeem, and swap actions, without requiring upfront liquidity checks. We formalized the behavior of our mechanism through an operational semantics, and we demonstrated that in certain scenarios, our LSM allows users to achieve their intents compared to traditional AMMs. Additionally, we defined a netting algorithm and proved properties regarding its complexity and behavior when handling deposit actions in relation to swap actions. We also discussed alternative design approaches for our mechanism that allow users to compute the netting off-chain and propose its result to the smart contract. This off-chain mechanism underscores the importance of incentive mechanisms in mitigating high gas fees and the associated costs of on-chain transactions. To validate our approach, we developed a simulator that we use to experiment with various application scenarios, providing valuable insights into the practical implications of our proposal.

In Chapter 4, we developed a formal model of intent-based proto-

cols in DeFi, focusing on their core design principles and operational dynamics. By analyzing representative protocols, we identified the key structural components that characterize intent-centric architectures, including intent aggregation, solver competition, and batch-based trade settlement. Our framework provides a unified and rigorous foundation for modeling these systems, laying the groundwork for implementing LSMs in decentralized environments. Building on this foundation, we formalized the semantics of intent-based protocols. The resulting model enables protocol-level reasoning about fairness, scalability, and MEV resistance, and serves as a design framework for developing more efficient, user-centric DeFi systems. In Chapter 5, we introduced a formal framework for specifying and enforcing security policies to regulate external calls in smart contracts. We enhanced a core calculus for smart contracts, by incorporating constructs that allow developers to specify their policies at the code level, and a run-time verification mechanism to check that the called code is compliant with the policy.

In conclusion, this thesis provides liquidity-efficient, user-aligned, and secure DeFi systems: a netting-style LSM adapted to AMMs, a formal semantics for intent protocols, and a policy calculus for safe external calls. Together, they demonstrate how to conserve liquidity through constraint-aware sequencing, aggregate it through intent-driven execution, and safeguard it with enforceable runtime policies. Taking all the contributions into account, they can interoperate: intents can settle on an LSM-style engine, and both layers can be hardened by policy enforcement. The result is a principled path toward markets that settle more with less capital lock-up, improved execution quality, and stronger resistance to manipulation. As DeFi continues to integrate with traditional rails improvement and expand cross-chains, the ideas developed in this thesis offer a durable foundation for protocols that are fair, scalable, MEV-resilient, and ready for real-world deployment.

A unifying methodological aspect of this thesis is the different use of transition systems tailored to the semantic requirements of each setting. In the netting-based LSM, the operational semantics emphasized execution ordering, atomicity, and liquidity constraints, enabling precise

reasoning about feasible transaction sequences and state-dependent execution. In contrast, the intent-based framework adopted labeled transition systems to model multi-agent interaction, competition, and coordination, supporting protocol-level analysis of fairness, scalability, and execution outcomes. Finally, the policy framework relied on execution-trace oriented semantics to monitor and enforce behavioral properties at runtime. These choices demonstrate how transition systems can be tailored to strike a balance between expressiveness, analyzability, and faithfulness to real-world DeFi mechanisms, rather than imposing a single, uniform formalism across heterogeneous protocol layers.

6.1 Future Work

Each contribution discussed in this thesis could be strengthened and extended in multiple ways.

For the first contribution, we plan to extend the current netting-based LSMs toward a more intent-centric perspective, in which user objectives are expressed directly and explicitly linked to execution outcomes. Rather than viewing netting as a cooperative mechanism for efficiency, we aim to formulate a unifying optimization framework that bridges cooperative efficiency and user-centric value. One promising direction is to model execution as the selection of a feasible subsequence of actions that maximizes a utility function derived from users' intents, subject to feasibility constraints such as balance preservation and the absence of overdrafts during execution. This would allow the model to reconcile collective efficiency with individual incentives within a single formal framework. Technically, this extension will involve combining the netting mechanism with competitive routing protocols to discover profitable execution paths across AMM pools and user networks. In parallel, we will align the LSM design with emerging blockchain and DeFi trends by refining the design choices discussed in Section 3.5 and by extending the incentive mechanisms introduced in Section 3.5.2. In particular, we plan to improve mechanisms that encourage participants to stake significant collateral in exchange for solving specific problems, with rewards dynamically ad-

justed based on the collateral ratio. We also intend to investigate whether transparency about the internal operation of the netting mechanism influences user behavior and potentially enables strategic manipulation by malicious actors. Finally, we will study the impact of queue length—a crucial system parameter—on the efficiency and responsiveness of the LSM, and evaluate how the mechanism behaves under different AMM models, including concentrated liquidity designs such as those introduced in Uniswap v3 [11].

We will align the LSM design, adapting our LSM to blockchain and DeFi trends, improve the design in Section 3.5, and enhance the design of incentive mechanisms introduced in Section 3.5.2, which encourage participants to stake significant collateral in exchange for solving specific problems, with rewards dynamically adjusted based on the collateral-to-bounty ratio. We are also interested in examining whether transparency regarding the internals of the netting mechanism influences user behavior, potentially leading to possible exploitation by malicious users. Furthermore, we will study how the queue length, a crucial parameter in our mechanism, impacts the efficiency of our LSM. Finally, another line of research involves considering how our mechanism work in different AMM models, such as Concentrated Liquidity which has been recently added to Uniswap v3 [11].

For the second contribution, we plan to extend the model with an auxiliary layer for cross-chain swaps that formalizes matchability across domains. We will also introduce a more flexible execution semantics for intents, supporting partial fills and the systematic generation of sub-intents, to better capture dynamic, real-world resolution workflows. We plan to formally analyse key properties of this enriched framework such as MEV resistance, fairness, and scalability, leveraging automated verification tools where appropriate. Finally, we aim to integrate the netting-style LSM from Chapter 3 to improve settlement efficiency and reduce on-chain overhead. These additions would bring the model closer to real-world conditions and further support the design of scalable, MEV-resilient, and economically efficient DeFi architectures.

For the third contribution, we plan to study how our mechanism can

be implemented. We define two possible directions: implementing the policy checking mechanism within the EVM or encoding it directly into Solidity. Moreover, we will explore other profitable scenarios within the DeFi ecosystem that could leverage our policy framework to prevent speculative actions. Finally, we will study if we can verify that an external call satisfies the policies up-front and the transaction without actually running it.

Appendix A

Supplementary Information

Acknowledgements

The last part of my PhD was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU through the fellowship "Modeling and security analysis of protocols for decentralized finance".

Declaration of generative AI and AI-assisted technologies in the writing process

During the preparation of this work, the author used Grammarly and ChatGPT for grammar and spelling checks. After using those tools/services, the author reviewed and edited the content as needed and takes full responsibility for the content of the publication.

Appendix B

DeFi Terminology

Terminology in DeFi evolves rapidly. Different protocols may label similar roles differently. The following definitions provide a consistent vocabulary for the thesis and align with contemporary protocol documentation and academic treatments of market microstructure and settlement.

Foundations & Assets

Digital Token A digital representation of value, rights, or access recorded on a blockchain.

Crypto-asset A digital representation of value or rights recorded on distributed ledger technology (DLT), transferable and storable electronically. Crypto-assets enable capital formation, alternative financing, and can function as payment instruments allowing cheaper and faster transfers with minimal intermediaries.

Stablecoin A crypto-asset designed to maintain a stable value by tracking a reference asset such as a fiat currency, commodity, or basket of assets.

USDC (USD Coin) A fiat-backed stablecoin pegged 1:1 to the U.S. dollar; issued by regulated entities with reserves held in cash and short-term U.S. dollar instruments.

DAI A crypto-collateralized stablecoin governed by MakerDAO that maintains a soft peg to the U.S. dollar through overcollateralized vaults, stability fees, and governance-controlled parameters.

ETH The native cryptocurrency of the Ethereum blockchain, used to pay transaction fees, stake for validation, and participate in decentralized applications.

Wallet Software or hardware that manages private keys and signs transactions to control blockchain accounts.

Smart Contract A self-executing piece of code deployed on a blockchain that automatically enforces predefined rules and agreements without intermediaries.

Protocol Tokens Governance or utility tokens that grant holders specific rights (e.g., voting or fee discounts) within a decentralized protocol.

Base Asset A comparatively safe and liquid asset that serves as a reference or foundation for pricing or settlement (e.g., stablecoins, fiat-pegged tokens, or sovereign assets).

Risk Asset A financial instrument or token characterized by high volatility and uncertain returns (e.g., equities, high-yield tokens, or volatile crypto pairs).

ERC Standards Ethereum Request for Comments; formal specifications defining interfaces and expected behaviors for Ethereum smart contracts (e.g., ERC-20, ERC-721).

ERC-20 An Ethereum token standard defining a uniform interface for fungible tokens, including transfer, approval, and balance-tracking functions.

ERC-721 An Ethereum token standard defining non-fungible tokens (NFTs), where each token has a unique identifier and distinct metadata. ERC-721 tokens are indivisible and non-interchangeable, making

them suitable for representing ownership of unique digital assets such as collectibles, artworks, or real-world assets.

Execution Context & Infrastructure

On-chain Executed directly on the blockchain, resulting in state changes that are publicly verifiable and immutable.

Off-chain Executed outside the blockchain environment (e.g., centralized servers or order books), with optional on-chain settlement.

Mempool A temporary storage area for unconfirmed transactions awaiting inclusion in a block. Validators prioritize higher-fee transactions, influencing confirmation speed.

Oracle A trusted service or decentralized mechanism that feeds external data (such as asset prices) into smart contracts for execution and settlement purposes.

Ethereum Virtual Machine (EVM) The runtime environment for executing smart contracts on Ethereum and compatible blockchains, ensuring deterministic behavior across all nodes.

Centralized Finance (CeFi) A financial ecosystem in which digital assets are managed and traded via centralized intermediaries (e.g., exchanges or custodians) who retain custody and control over user funds.

Decentralized Finance (DeFi) A blockchain-based financial ecosystem that replaces centralized intermediaries with smart contracts to enable permissionless, non-custodial services such as trading, lending, and staking.

Third Trusted Party (TTP) An intermediary entity relied upon to facilitate trust, authentication, or settlement between transacting parties.

Markets, AMMs & Liquidity

Decentralized Exchange (DEX) A peer-to-peer marketplace enabling users to trade directly without intermediaries. DEXs are non-custodial and rely on smart contracts for trade execution and settlement.

Automated Market Maker (AMM) A DEX design that replaces order books with liquidity pools, where prices are determined algorithmically by mathematical formulas (e.g., constant product, stable-swap).

Constant Function Market Maker (CFMM) A class of AMMs where trades preserve a constant mathematical invariant (e.g., Uniswap's $x \times y = k$), ensuring continuous liquidity and deterministic pricing.

Private Market Maker (PMM) A specialized, privately-operated liquidity provider that uses custom pricing algorithms and executes a hybrid model of off-chain quotation and on-chain settlement. PMMs offer Request for Quote (RFQ) style services, aggregate both on-chain and off-chain liquidity, and enable users to maintain custody of their assets while achieving improved pricing and lower slippage compared to traditional AMMs.

Liquidity Pool A pool of two or more tokens used to facilitate swaps under an AMM curve and to accrue trading fees.

Liquidity Provider (LP) A user who deposits assets into a liquidity pool in exchange for a proportional share of pool fees and LP tokens.

Liquidity Provider (LP) Tokens Tokens issued to liquidity providers representing their proportional share of the pool's reserves and earned fees. LP tokens are transferable and may be used as collateral or staked in other protocols.

Add/Remove Liquidity The process of depositing or withdrawing assets to or from a liquidity pool, minting or burning LP tokens accordingly.

Concentrated Liquidity An AMM mechanism (introduced in Uniswap v3) that allows LPs to allocate capital within specific price ranges, increasing efficiency but introducing range-specific risks.

Slippage The difference between a trade's expected price and its actual execution price, typically increasing with order size, volatility, or limited liquidity.

Thin Liquidity A market condition where limited depth causes small trades to move prices significantly, increasing volatility and slippage.

Impermanent Loss The loss experienced by liquidity providers when the relative prices of pooled tokens diverge from their initial ratio. The effect may reverse if prices revert or be offset by trading fees.

Aggregator A routing service that searches across liquidity venues to minimize slippage, improve execution rates, and reduce transaction costs.

Total Value Locked (TVL) The total value of assets deposited within a protocol, often used as a measure of liquidity and ecosystem size.

Gas Fee The payment to validators for transaction execution on Ethereum, varying with network congestion.

Fee A cost charged by protocols, validators, or liquidity pools for providing services such as transaction processing, trading, or liquidity provision.

Reward Compensation or yield distributed to users (e.g., liquidity providers or validators) for contributing resources to a network or protocol.

Volatility The degree of variation in an asset's price over time, used as a measure of market risk or uncertainty.

Pullback A temporary decline or correction in the price of an asset following a sustained upward trend, often viewed as a short-term market adjustment.

Trading Primitives & Order Flow

Swap The exchange of one token for another against an AMM pool at a price determined by the pool's reserves, fees, and slippage.

Flash Loan An uncollateralized loan borrowed and repaid within the same transaction; if repayment fails, the transaction reverts.

Flash Swap (*Uniswap v2*) A mechanism that allows users to receive tokens before paying for them within a single transaction, reverting if the equivalent value is not returned by the end.

Order Book A list of active buy (bid) and sell (ask) orders arranged by price, used to determine the best available execution.

Limit Order A trade instruction to execute at a specified price or better; it provides liquidity if not immediately filled.

Market Order A trade instruction to execute immediately at the best available price; it consumes liquidity and may incur slippage.

Request for Quote (RFQ) A mechanism by which a user requests price quotes from one or more market makers, selecting and settling the most favorable quote on-chain.

Order Flow Auction A competitive auction where solvers or executors bid to fulfill user orders or intents, typically optimizing for price or efficiency.

Intent A signed statement of a desired outcome, specifying constraints such as minimum output or time-to-live, allowing solvers to compete for execution.

Solver An agent that finds and executes an optimal strategy to satisfy a user's intent, potentially aggregating liquidity across venues.

Bot An automated agent that monitors markets or mempools and submits transactions or intents based on predefined conditions or arbitrage opportunities.

Coincidence of Wants (CoW) An economic condition in which two or more parties each possess assets that the other desires, enabling a direct exchange without intermediaries. In DeFi (e.g., CoW Protocol), it refers to matching complementary buy and sell orders so that trades can be settled peer-to-peer without relying on external liquidity pools or MM.

Tranching The splitting of a single order into smaller slices that resolvers can fill independently (e.g., during a Dutch auction as the price decays), enabling partial fills, greater competition, and reduced slippage.

Maximal Extractable Value (MEV) The additional profit that can be captured by reordering, inserting, or censoring transactions within a block (e.g., through arbitrage or sandwich attacks).

Interoperability

Bridge Infrastructure for transferring assets or messages between different blockchains using mechanisms such as lock-and-mint, burn-and-release, or light-client proofs.

Cross-Chain Swap A trade that spans multiple blockchains, typically coordinated via bridging or messaging protocols.

Cross-Chain Interoperability Protocol (CCIP) A protocol that facilitates secure communication and asset transfers between blockchains using standardized interfaces and decentralized oracle networks.

Governance & Custody

Decentralized Autonomous Organization (DAO) An on-chain governance system that collectively manages protocol decisions, upgrades, and treasury allocations through token-holder voting.

Multisig Wallet A wallet requiring multiple signatures to authorize a transaction, enhancing security and shared control.

Consensus & Validation

Validator A network participant who stakes assets to propose and verify new blocks, earning rewards for honest participation and facing penalties for misbehavior.

Proof of Stake (PoS) A consensus algorithm that selects validators based on the size and duration of their stake, providing energy-efficient security and transaction finality.

Security & Attack Vectors

Direct Price Manipulation (DPM) Attack An exploit where an attacker temporarily distorts an AMM or oracle price (e.g., via large trades or flash loans) to profit from arbitrage, liquidations, or mispriced collateral.

Indirect Price Manipulation (IPM) Attack An exploit that manipulates secondary mechanisms, such as oracle feeds or lending pool valuations, without directly altering AMM prices.

Reentrancy A smart contract vulnerability where an attacker repeatedly calls a function before the initial execution completes, allowing unauthorized withdrawals or state changes.

Payments, Settlement, and LSM

Interbank Payments Transfers of funds between banks to settle obligations arising from payment systems, securities transactions, or other financial activities.

Real-Time Gross Settlement (RTGS) An electronic payment system in which interbank transactions are processed and settled individually, in real time. Each payment is irrevocable once executed, ensuring immediate transfer of funds between participating institutions.

Continuous Net Settlement (CNS) A settlement process operated by the National Securities Clearing Corporation (NSCC) for the clearing and settlement of securities transactions. CNS continuously nets trades among participants, reducing the number of securities and cash movements required. Its main advantage is the minimization of exchanges between counterparties, thereby improving efficiency and lowering settlement risk.

Stunting State A temporary operating condition where queued payments are held back to preserve liquidity or await offsetting opportunities; released once buffers are restored or viable netting sets identified.

Deadlock A settlement delay arising when liquidity is insufficient to process any queued payments.

Gridlock A condition in which queued payments mutually block one another despite sufficient aggregate liquidity.

Gridlock Resolution Algorithm A multilateral netting procedure that searches for offsetting cycles or compatible sets of queued payments and selects a subset that can settle simultaneously (maximizing feasible settlement subject to balances and limits) without requiring additional liquidity.

Liquidity-Saving Mechanism (LSM) A hybrid RTGS feature combining queuing and partial multilateral netting to reduce intraday liquidity needs and resolve gridlock.

Netting Bilateral or multilateral offsetting of queued obligations to reduce gross payment flows into smaller net positions within an LSM.

Bibliography

[1] Kaihua Qin et al. "CeFi vs. DeFi–Comparing Centralized to Decentralized Finance". In: *arXiv preprint arXiv:2106.08157* (2021).

[2] Sam Werner et al. "Sok: Decentralized finance (defi)". In: *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*. 2022, pp. 30–46.

[3] Jiahua Xu et al. "Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols". In: *ACM Computing Surveys* 55.11 (2023), pp. 1–50.

[4] Antoine Martin and James McAndrews. "Liquidity-saving mechanisms". In: *Journal of Monetary Economics* 55.3 (2008), pp. 554–567.

[5] Tarun Chitra et al. "An analysis of intent-based markets". In: *arXiv preprint arXiv:2403.02525* (2024).

[6] Yash Chandak. *Under the Hood of Intent-Based Bridges. Should you use intent-based bridges? A guide to the pros and cons.* LI.FI Knowledge Hub. July 2, 2024. URL: https://li.fi/knowledge-hub/under-the-hood-of-intent-based-bridges/ (visited on 09/27/2025).

[7] Uniswap. *Uniswap V1 Documentation — Overview*. https://docs.uniswap.org/contracts/v1/overview. Accessed: January 14, 2026. 2019.

[8] Anonymous Person. *Sushi protocol documentation*. https://docs.sushi.com/pdf/whitepaper.pdf. Accessed: April, 2024. 2020.

[9] PancakeSwap Labs. *PancakeSwap Documentation*. https://docs.pancakeswap.finance/. Accessed: January 14, 2026. 2025.

[10] Balancer Labs. *Balancer Protocol Whitepaper*. https://docs.balancer.fi/whitepaper.pdf. Accessed: January 14, 2026. 2025.

[11] Adams Hayden et al. *Uniswap V3*. https://uniswap.org/whitepaper-v3.pdf. Accessed: April, 2024. 2021.

[12] Michael Egorov. "Stableswap-efficient mechanism for stablecoin liquidity". In: *Retrieved Feb* 24 (2019), p. 2021.

[13] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. "Maximizing extractable value from automated market makers". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2022, pp. 3–19.

[14] Kaihua Qin et al. "Attacking the defi ecosystem with flash loans for fun and profit". In: *International conference on financial cryptography and data security*. Springer. 2021, pp. 3–32.

[15] Aave Documentation V3. *Flash Loans*. https://docs.aave.com/developers/guides/flash-loans. Accessed: August, 2024. 2023.

[16] Kaihua Qin et al. "Attacking the defi ecosystem with flash loans for fun and profit". In: *International conference on financial cryptography and data security*. Springer. 2021, pp. 3–32.

[17] Hitoshi Hayakawa. "How a liquidity saving mechanism affects bank behavior in interconnected payment networks". In: *Journal of Economic Interaction and Coordination* 20.1 (2025), pp. 41–71.

[18] Bank of Canada and Monetary Authority of Singapore. *Enabling Cross-Border High Value Transfer Using Distributed Ledger Technologies*. https://www.mas.gov.sg/-/media/Jasper-Ubin-Design-Paper.pdf. Accessed: July, 2024. 2018.

[19] Michael M Güntzer, Dieter Jungnickel, and Matthias Leclerc. "Efficient algorithms for the clearing of interbank payments". In: *European Journal of Operational Research* 106.1 (1998), pp. 212–219.

[20] Morten Bech and Kimmo Soramaki. "Gridlock Resolution in Payment Systems". In: *Danmarks Nationalbank Monetary Review* (July 2001), pp. 67–79. DOI: 10.2139/ssrn.274290.

[21] Shengjiao Cao et al. "Decentralized privacy-preserving netting protocol on blockchain for payment systems". In: *Financial Cryptography and Data Security: 24th International Conference, FC 2020*. Springer. 2020, pp. 137–155.

[22] Aleksander Junge. *Liquidity saving mechanism for AMMs inspired by netting*. https://github.com/aleksanderJunge/AMM-netting-inspired-protocol. Accessed: October, 2024. 2024.

[23] Margherita Renieri et al. "A Netting Protocol for Liquidity-saving Automated Market Makers". In: *Proceedings of the 6th Distributed Ledger Technology Workshop (DLT 2024)*. Vol. 3791. CEUR Workshop Proceedings. Turin, Italy, 2024. URL: https://ceur-ws.org/Vol-3791/.

[24] Margherita Renieri et al. "Netting-based Liquidity-saving Automated Market Makers". In: *Blockchain: Research and Applications* (2025). DOI: 10.1016/j.bcra.2025.100361.

[25] *Flash Loans*. https://docs.aave.com/faq/flash-loans. Accessed: July, 2024. 2021.

[26] *Flash Swaps*. https://docs.uniswap.org/contracts/v2/guides/smart-contract-integration/using-flash-swaps. Accessed: July, 2024. 2023.

[27] CoW Protocol Developers. *CoW Protocol Documentation*. https://docs.cow.fi/. Accessed: June, 2025. 2021.

[28] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. "A theory of Automated Market Makers in DeFi". In: *Logical Methods in Computer Science* 18.4 (2022), p. 12.

[29] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch Lafuente. "Maximizing extractable value from automated market makers". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2022, pp. 3–19.

[30] Ken Naganuma et al. "Decentralized Netting Protocol over Consortium Blockchain". In: *International Symposium on Information Theory and Its Applications,ISITA 2018*. IEEE, 2018, pp. 174–177. DOI: 10.23919/ISITA.2018.8664259.

[31] Xin Wang et al. "Inter-Bank Payment System on Enterprise Blockchain Platform". In: *11th IEEE International Conference on Cloud Computing, CLOUD 2018*. IEEE Computer Society, 2018, pp. 614–621. DOI: 10.1109/CLOUD.2018.00085.

[32] The Mangrove Team. *Mangrove V5 Whitepaper*. https://whitepaper.mangrove.exchange/. Accessed: July, 2024. 2021.

[33] Guillermo Angeris et al. “Optimal routing for constant function market makers”. In: *Proceedings of the 23rd ACM Conference on Economics and Computation*. 2022, pp. 115–128.

[34] Theo Diamandis et al. “An Efficient Algorithm for Optimal Routing Through Constant Function Market Makers”. In: *arXiv preprint arXiv:2302.04938* (2023).

[35] Vincent Danos, Hamza El Khalloufi, and Julien Prat. “Global order routing on exchange networks”. In: *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC*. Springer. 2021, pp. 207–226.

[36] *UniswapX*. https://uniswap.org/whitepaper-uniswapx.pdf. Accessed: July, 2024. 2023.

[37] *Uniswap V4*. https://github.com/Uniswap/v4-core/blob/main/docs/whitepaper-v4.pdf. Accessed: July, 2024. 2023.

[38] Philip Daian et al. “Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability”. In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 910–927. DOI: 10.1109/SP40000.2020.00040. URL: https://doi.org/10.1109/SP40000.2020.00040.

[39] Massimo Bartoletti, James Hsin-yu Chiang, and Alberto Lluch-Lafuente. “Maximizing Extractable Value from Automated Market Makers”. In: *Financial Cryptography and Data Security - 26th International Conference, FC 2022, Grenada, May 2-6, 2022, Revised Selected Papers*. Ed. by Ittay Eyal and Juan A. Garay. Vol. 13411. Lecture Notes in Computer Science. Springer, 2022, pp. 3–19. DOI: 10.1007/978-3-031-18283-9_1. URL: https://doi.org/10.1007/978-3-031-18283-9%5C_1.

[40] Frederik Lührs et al. *The Road Towards an Encrypted Mempool on Ethereum*. Accessed: April 10, 2025. 2024. URL: https://docs.shutter.network/docs/shutter/research/the_road_towards_an_encrypted_mempool_on_ethereum.

[41] Flashbots. *Flashbots Documentation*. https://docs.flashbots.net/. Accessed: September, 2024. 2024.

[42] Flashbots Collective. *Flashbots Transparency Report: MEV-Share, SGX block building & ETHDenver - The Flashbots Ship*. https://collective.flashbots.net. Accessed: September, 2024. 2023.

[43] John Doe and Jane Smith. "Decentralized Lending Protocols: Efficiency and Risk Management". In: *Journal of Blockchain Finance* 15.4 (2023), pp. 100–120. DOI: 10.1234/jbf.2023.001.

[44] Christopher Goes, Awa Sun Yin, and Adrian Brink. "Anoma: a unified architecture for full-stack decentralised applications". In: *Anoma Research Topics, Aug* (2023).

[45] Anton Bukov et al. *1inch Fusion+: Intent-Based Atomic Cross-Chain Swaps*. White paper. Accessed 2025-09-01. 1inch Network, Sept. 2024. URL: https://1inch.io/assets/1inch-fusion-plus.pdf (visited on 09/01/2025).

[46] 0x Labs. *Matcha by 0x*. https://matcha.xyz/. Accessed: June, 2025. 2024.

[47] Mengqian Zhang et al. "Maximal Extractable Value in Batch Auctions". In: *Proceedings of the 26th ACM Conference on Economics and Computation*. New York, NY, USA: Association for Computing Machinery, 2025, p. 510. ISBN: 9798400719431.

[48] Eric Budish, Peter Cramton, and John Shim. "The high-frequency trading arms race: Frequent batch auctions as a market design response". In: *The Quarterly Journal of Economics* 130.4 (2015), pp. 1547–1621.

[49] CoW DAO. *How CoW Swap Solves the MEV Problem*. Accessed 2025-09-01. CoW DAO. URL: https://cow.fi/learn/how-cow-swap-solves-the-mev-problem (visited on 09/01/2025).

[50] Maurice Herlihy. "Atomic cross-chain swaps". In: *Proceedings of the 2018 ACM symposium on principles of distributed computing*. 2018, pp. 245–254.

[51] Hashflow Protocol Developers. *Hashflow Documentation*. https://docs.hashflow.com/hashflow. Last accessed June, 2025. 2025.

[52] Mark Toda, Matt Rice, and Nick Pai. *ERC-7683: Cross Chain Intents*. https://eips.ethereum.org/EIPS/eip-7683. Ethereum Improvement Proposal; Standards Track: ERC; status: Draft. Created 2024-04-11. Accessed 2025-09-02.

[53] Stephen Monn and Bikem Bengisu. *ERC-7521: General Intents for Smart Contract Wallets*. https://eips.ethereum.org/EIPS/eip-7521. Ethereum Improvement Proposal; Standards Track: ERC; status: Draft. Created 2023-09-19. Accessed 2025-09-02.

[54] Suryansh Shrivastava. *ERC 7521 Core Concepts Explainer*. *Block Magnates* (blog). Published June 16, 2024. Accessed Sept. 30, 2025. June 2024. URL: https://blog.blockmagnates.com/erc-7521-core-concepts-explainer-8f5b47751e45.

[55] Daniel Liberto. *Over-the-Counter (OTC) Markets: Trading and Securities*. Fact checked by Stella Osoba. Investopedia. Mar. 6, 2025. URL: https://www.investopedia.com/terms/o/otc.asp (visited on 09/02/2025).

[56] Ciamac C Moallemi and Dan Robinson. "Loss-Versus-Fair: Efficiency of Dutch Auctions on Blockchains". In: *arXiv preprint arXiv:2406.00113* (2024).

[57] Wenqing Li et al. "Towards Blockchain Interoperability: A Comprehensive Survey on Cross-Chain Solutions". In: *Blockchain: Research and Applications* (2025), p. 100286.

[58] *Chainlink CCIP — Cross-Chain Interoperability Protocol*. 2025. URL: https://docs.chain.link/ccip (visited on 09/30/2025).

[59] *AggLayer Documentation*. 2025. URL: https://docs.agglayer.dev/ (visited on 09/30/2025).

[60] Kyber Network. *KyberSwap Aggregator: Overview*. 2024. URL: https://docs.kyberswap.com/kyberswap-solutions/kyberswap-aggregator (visited on 08/27/2025).

[61] *NEAR Intents Documentation*. https://docs.near-intents.org/near-intents/. Accessed: October 2, 2025. 2025.

[62] ODOS. *Odos Documentation Portal: Introduction*. 2025. URL: https://docs.odos.xyz/home/about (visited on 09/04/2025).

[63] *Odos Documentation — Cross-Chain*. https://docs.odos.xyz/use/crosschain. Accessed: October 2, 2025. 2025.

[64] Andrea Canidio and Felix Henneke. *Fair Combinatorial Auction for Blockchain Trade Intents: Being Fair without Knowing What is Fair*. 2024. arXiv: 2408.12225 [econ.TH]. URL: https://arxiv.org/abs/2408.12225.

[65] Velora Team. *Velora Delta API*. https://developers.velora.xyz/api/velora-api/velora-delta-api. Accessed 2025-09-04. 2025.

[66] Velora Team. *Velora Market API*. https://developers.velora.xyz/api/velora-api/velora-market-api. Accessed 2025-09-04. 2025.

[67] Across Protocol Team. *Across Documentation*. https://docs.across.to/. Accessed 2025-09-03. 2025.

[68] Remco Bloemen, Leonid Logvinov, and Jacob Evans. *EIP-712: Typed Structured Data Hashing and Signing*. 2017. URL: https://eips.ethereum.org/EIPS/eip-712 (visited on 10/07/2025).

[69] Conor McMenamin et al. "Fairtradex: A decentralised exchange preventing value extraction". In: *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*. 2022, pp. 39–46.

[70] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. "Foundations and tools for the static analysis of ethereum smart contracts". In: *International conference on computer aided verification*. Springer. 2018, pp. 51–78.

[71] OWASP Project. *Reentrancy attack*. https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC01-reentrancy-attacks.html. Accessed: August, 2024. 2024.

[72] John Fáwọlé. *A Broad Overview of Reentrancy Attacks in Solidity Contracts*. https://www.quicknode.com/guides/ethereum-development/smart-contracts/a-broad-overview-of-reentrancy-attacks-in-solidity-contracts. Accessed: August, 2024. 2023.

[73] Fred B Schneider. "Enforceable security policies". In: *ACM Transactions on Information and System Security (TISSEC)* 3.1 (2000), pp. 30–50.

[74] Dexter Kozen. "Language-Based Security: Invited Lecture". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1999, pp. 284–298.

[75] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. "A minimal core calculus for Solidity contracts". In: *Data Privacy Management, Cryptocurrencies and Blockchain Technology: ESORICS 2019 International Workshops, DPM 2019 and CBT 2019, Proceedings*. Springer. 2019, pp. 233–243.

[76] Margherita Renieri and Letterip Galletta. "A Policy Framework for Regulating External Calls in Smart Contracts". In: *Software Engineering and Formal Methods*. Ed. by A. Madeira and A. Knapp. Springer Nature Switzerland, 2025. DOI: 10.1007/978-3-031-77382-2_4.

[77] FailSafe. *FailSafe Web3 Security Report 2025*. https://getfailsafe.com/failsafe-web3-security-report-2025/. Accessed: July, 2025. 2025.

[78] Stefanos Chaliasos et al. "Smart contract and defi security tools: Do they meet the needs of practitioners?" In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–13.

[79] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. "A survey of attacks on ethereum smart contracts (sok)". In: *International conference on principles of security and trust*. Springer. 2017, pp. 164–186.

[80] U.S. Securities and Exchange Commission. *Report of Investigation Pursuant to Section 21(a) of the Securities Exchange Act of 1934: The DAO. U.S. Securities and Exchange Commission Report No. 34-81207*. https://www.sec.gov/files/litigation/investreport/34-81207.pdf. 2017.

[81] Hellis Tamm and Margus Veanes. "Theoretical aspects of symbolic automata". In: *International Conference on Current Trends in Theory and Practice of Informatics*. Springer. 2017, pp. 428–441.

[82] Ken Thompson. "Programming techniques: Regular expression search algorithm". In: *Communications of the ACM* 11.6 (1968), pp. 419–422.

[83] Bruce William Watson. *A taxonomy of finite automata construction algorithms*. English. Computing science notes. Technische Universiteit Eindhoven, 1993.

[84] Ethereum.org. *ERC-721: Non-Fungible Token Standard*. https://ethereum.org/en/developers/docs/standards/tokens/erc-721/. Accessed: 2025-08-26.

[85] Jason Milionis, Ciamac C Moallemi, and Tim Roughgarden. "Automated market making and arbitrage profits in the presence of fees". In: *arXiv preprint arXiv:2305.14604* (2023).

[86] Ningyu He Bosi Zhang et al. “Following Devils’ Footprint: Towards Real-time Detection of Price Manipulation Attacks”. In: (2025).

[87] PeckShield. *Cheese Bank Incident: Root Cause Analysis*. Nov. 2020. URL: https://peckshield.medium.com/cheese-bank-incident-root-cause-analysis-d076bf87a1e7 (visited on 08/20/2025).

[88] *Cheese — Official Whitepaper*. https://resources.cryptocompare.com/asset-management/7064/1723025046359.pdf. Accessed: October 1, 2025.

[89] Samuel Drews and Loris D’Antoni. “Learning symbolic automata”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2017, pp. 173–189.

[90] Dana Fisman, Hadar Frenkel, and Sandra Zilles. “Inferring symbolic automata”. In: *Logical Methods in Computer Science* 19 (2023).

[91] Benjamin Eriksson et al. “Black Ostrich: Web application scanning with string solvers”. In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 549–563.

[92] Christian Skalka and Scott Smith. “History effects and verification”. In: *Asian Symposium on Programming Languages and Systems*. Springer. 2004, pp. 107–128.

[93] Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. “History-based access control with local policies”. In: *Proceedings of 8th International Conference on Foundations of Software Science and Computational Structures, FOSSACS 2005.* Springer. 2005, pp. 316–332.

[94] Massimo Bartoletti et al. “Local policies for resource usage analysis”. In: *ACM Trans. Program. Lang. Syst.* 31.6 (2009), 23:1–23:43.

[95] Clara Schneidewind et al. “ethor: Practical and provably sound static analysis of ethereum smart contracts”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 2020, pp. 621–640.

[96] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. “A Semantic Framework for the Security Analysis of Ethereum Smart Contracts”. In: *POST*. Vol. 10804. LNCS. Springer, 2018, pp. 243–269. DOI: 10.1007/978-3-319-89722-6_10.

[97] Siwei Wu et al. “DeFiRanger: Detecting DeFi Price Manipulation Attacks”. In: *IEEE Transactions on Dependable and Secure Computing* PP (2023), pp. 1–15. DOI: 10.1109/TDSC.2023.3346888.

[98] Dabao Wang et al. “Towards a first step to understand flash loan and its applications in defi ecosystem”. In: *Proceedings of the Ninth International Workshop on Security in Blockchain and Cloud Computing*. 2021, pp. 23–28.

[99] Queping Kong et al. “DeFiTainter: Detecting Price Manipulation Vulnerabilities in DeFi Protocols”. In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1144–1156.

[100] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. “Flashsyn: Flash loan attack synthesis via counter example driven approximation”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024, pp. 1–13.

[101] Bin Wang et al. “Blockeye: Hunting for defi attacks on blockchain”. In: *2021 IEEE/ACM 43rd international conference on software engineering: companion proceedings (ICSE-companion)*. IEEE. 2021, pp. 17–20.

[102] Xinyao Xu et al. “Quantitative Runtime Monitoring of Ethereum Transaction Attacks”. In: *Proceedings of the ACM on Web Conference 2025*. WWW ’25. Sydney NSW, Australia: Association for Computing Machinery, 2025, pp. 4146–4159. ISBN: 9798400712746. DOI: 10.1145/3696410.3714682. URL: https://doi.org/10.1145/3696410.3714682.

[103] Liyi Zhou et al. “On the just-in-time discovery of profit-generating transactions in defi protocols”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 919–936.