DeFi vulnerabilities in the eUTxO model

Georgios Tsoumas
7115142200018

Examination committee:
Aris Pagourtzis, School of Electrical and Computer Engineering, National Technical University of Athens.
Dimitris Fotakis, School of Electrical and Computer Engineering, National Technical University of Athens.
Nikos Leonardos, School of Electrical and Computer Engineering, National Technical University of Athens.

Supervisor:
Aris Pagourtzis, Professor,
School of Electrical and Computer Engineering,
National Technical University of Athens.

ABSTRACT

Miner Extractable Value (MEV) refers to the additional profits block producers can earn by manipulating the inclusion and ordering of transactions within a block. This practice, known as transaction reordering attacks, typically targets *Decentralized Finance* (DeFi) transactions. MEV incidents alone cost the Ethereum ecosystem over $350 million in 2023.

MEV opportunities are specific to the underlying blockchain infrastructure, including network topology, smart contract logic, and accounting model. While MEV in Ethereum has been well-studied in the academic literature, there has been little research on MEV in the eUTxO model.

In this work, we present the necessary background knowledge regarding MEV in the eUTxO model. We compare the account-based and eUTxO accounting models in the context of MEV. Additionally, we introduce the major *Decentralized Exchange* (DEX) paradigms, namely Constant Product Market Makers and Limit Order Books. Finally, we offer a game-theoretic definition of MEV and analyze three vulnerabilities within Cardano's eUTxO DeFi ecosystem.

ΣΥΝΟΨΗ

Η *Εξαγώγιμη Αξία από Παραγωγούς Μπλοκ* (MEV) αναφέρεται στα επιπλέον κέρδη που μπορούν να αποκομίσουν οι παραγωγοί μπλοκ μέσω της χειραγώγησης της εισαγωγής και της σειράς εκτέλεσης των συναλλαγών σε ένα μπλοκ. Αυτή η πρακτική, γνωστή ως επιθέσεις αναδιάταξης συναλλαγών, στοχεύει κυρίως συναλλαγές των *Αποκεντρωμένων Χρηματοοικονομικών* (DeFi). Μόνο το 2023, τα περιστατικά MEV κόστισαν στο οικοσύστημα του Ethereum πάνω από 350 εκατομμύρια δολάρια.

Οι ευκαιρίες MEV είναι συγκεκριμένες ως προς την υποκείμενη υποδομή του εκάστοτε blockchain, περιλαμβάνοντας την τοπολογία του δικτύου, τη λογική των έξυπνων συμβολαίων και το λογιστικό μοντέλο. Αν και το MEV στο Ethereum έχει μελετηθεί εκτενώς στη βιβλιογραφία, η έρευνα σχετικά με το MEV στο μοντέλο eUTxO είναι περιορισμένη.

Σε αυτή την εργασία, παρουσιάζουμε τις απαραίτητες γνώσεις σχετικά με το MEV στο μοντέλο eUTxO. Συγκρίνουμε τα λογιστικά μοντέλα λογαριασμών και eUTxO σε σχέση με το MEV. Επιπλέον, εισάγουμε τα κύρια παραδείγματα *Αποκεντρωμένων Ανταλλακτηρίων* (DEX), συγκεκριμένα τις αυτόματες αγορές σταθερού γινομένου και τα βιβλία παραγγελιών ορίου. Τέλος, προσφέρουμε έναν παιγνιοθεωρητικό ορισμό του MEV και αναλύουμε τρεις ευπάθειες στο οικοσύστημα DeFi του Cardano με το μοντέλο eUTxO.

CONTENTS

1 Introduction **1**

2 Preliminaries **5**
- 2.1 Cryptography 5
- 2.2 Game theory 6
  - 2.2.1 Game theory in Blockchain; Classical results 7
- 2.3 The Cardano blockchain 8
  - 2.3.1 Network Layer 8
  - 2.3.2 Consensus layer 9
  - 2.3.3 Application Layer 10

3 Ledger Models **13**
- 3.1 The account-based model 13
  - 3.1.1 Ethereum Transactions 14
- 3.2 The UTxO model 17
  - 3.2.1 UTxO Transactions 19
  - 3.2.2 Bitcoin Transactions 19
- 3.3 The eUTxO model 21
  - 3.3.1 eUTxO Transactions 22
  - 3.3.2 Concurrency 23
- 3.4 eUTxO and account-based comparison 24

4 Decentralized Finance **29**
- 4.1 Types of Security 30
- 4.2 Limit Order Books 32
- 4.3 Automatic Market Makers 34
  - 4.3.1 Constant Product Automatic Market Makers 36
  - 4.3.2 UniSwap 40
- 4.4 LOB and CPMM comparison 40

5 Transaction Ordering **43**
- 5.1 Rational Block Producers 44
- 5.2 Transaction Reordering Attacks 45

5.2.1 Sandwich Attacks . . . 45
5.2.2 Cyclic Arbitrage Opportunities . . . 46
5.2.3 Front-running in the eUTxO Model . . . 47
5.3 Block Producer Surplus . . . 48
5.3.1 Off-Chain Agreement . . . 49

6 eUTxO DeFi **51**
6.1 Transaction Batchers . . . 51
6.2 MuesliSwap . . . 53
6.2.1 BPS incidents in MuesliSwap . . . 54
6.3 MinSwap . . . 56
6.3.1 Sandwich attacks in MinSwap . . . 57
6.4 SundaeSwap . . . 58
6.4.1 Fatal front-running SundaeSwap batch transactions . . . 59

7 Conclusion **63**
7.1 Future Work . . . 63

Bibliography **64**

CHAPTER 1

INTRODUCTION

In traditional digital finance, if Alice wants to transact with Bob and exchange value between them, a third entity usually has to step in and facilitate that transaction. This creates additional complexities for Alice, namely:

- Increased costs in the form of banking fees.
- Privacy concerns. Both Alice and Bob need to share much of their financial and personal information with the bank.
- Counterparty risk. The bank could default or deny facilitating the transaction for various reasons, such as compliance or political beliefs.
- Geographical constraints. Many third-world countries do not have access to the banking system.

Overall, the traditional banking system has several weaknesses. For the past decades, many have advocated for open, confidential, and censorship-resistant payment systems. In other words, a *decentralized permissionless transaction ledger* that anyone can use and that cannot be manipulated by any central authority.

In 2008, Satoshi Nakamoto introduced Bitcoin [1], an electronic peer-to-peer payment system. Bitcoin's novel idea is the Nakamoto-style consensus, a new consensus mechanism. This, paired with Proof of Work [2], solved the permissionless decentralized ledger problem, as formally proven by GKL in [3]. Bitcoin is a protocol that allows users to transact with one another using pseudonyms. Several entities known as *miners* or *block producers* (**BPs**) are tasked with the maintenance of the decentralized transaction ledger by participating in Bitcoin's consensus mechanism. They also receive monetary rewards denominated in BTC (Bitcoin's native token), thus encouraging honest participation and constant uptime. Due to Bitcoin's use of cryptography, all protocols that facilitate payments through a permissionless transaction ledger have come to be known as cryptocurrencies.

Although Bitcoin solved the problem of online decentralized payments, there were still many financial services that it could not provide, such as loans, contracts, and asset

ownership. Additionally, it was apparent since the early days of Bitcoin that it could not scale sufficiently to accommodate the financial needs of the entire planet.

Bitcoin's launch opened the floodgates for many other projects, all promising to fulfill its original goal. The first notable cryptocurrency to follow through though, was Ethereum [4], launching in July 30, 2015. Ethereum's novelty, originally envisioned by Vitalik Buterin, was to imbue a blockchain with additional programmability, thus enabling the blockchain to finalize a payment only when certain conditions are met, or issuing an asset on top of the blockchain. This notion was named *smart contract*, due their ability to automatically enforce and execute agreements written in code, functioning like traditional contracts but without the need for intermediaries.

Now enabled by smart contracts, users can issue and maintain secondary assets (i.e., other than the blockchain's native token). The most common utilities of these assets are:

- **Utility tokens**: They provide users with access to a product or service within a blockchain ecosystem.
- **Governance tokens**: They grant holders the right to participate in decision-making processes.
- **Stablecoins**: They are pegged to a stable asset, often a fiat currency like the US dollar.

These assets are collectively known as fungible tokens, non-native tokens, or cryptoassets. With the mass adoption of these tokens, the need for a way to trade them quickly arose. Centralized exchanges (**CEX**s) proved to be untrustworthy, with many CEXs, such as Mt. Gox[1] and QuadrigaCX[2], collapsing due to mismanagement or technical breaches in security.

This gave rise to the concept of a *Decentralized Exchange* (**DEX**), which is an exchange that operates by leveraging blockchain infrastructure and lacks a central owner. Building on an idea that was already advocated by many, including Vitalik Buterin [5], the UniSwap [6] DEX launched in November 2, 2018. UniSwap, an Ethereum-based DEX, attracted a significant user base and achieved market dominance, becoming the benchmark for all other DEXs[3]. We will analyze DEX primitives in detail in the following chapters.

Following Ethereum's massive success, many smart contract-capable blockchains were launched, including Cardano in 2017. Cardano, originally envisioned by Ethereum co-founder Charles Hoskinson [7], is a Proof of Stake blockchain designed to support smart contracts while inheriting key aspects of Bitcoin's original design, such as Nakamoto-style consensus and the UTxO accounting model. The Cardano team succeeded in enhancing Satoshi's UTxO model with smart contract capabilities, naming this novel approach the *extended Unspent Transaction Outputs* (eUTxO) model.

[1] The Guardian's report: https://www.theguardian.com/technology/2017/jul/11/gox-bitcoin-exchange-mark-karpeles-on-trial-japan-embezzlement-loss-of-millions

[2] Reuters' report: https://www.reuters.com/article/us-crypto-currencies-quadriga/canadian-cryptocurrency-firm-collapsed-due-to-ponzi-scheme-by-late-founder-regulator-says-idUSKBN23I3AF/

[3] As of the time of writing, the total value locked in UniSwap is $3.74 billion: https://defillama.com/chain/Ethereum

Cardano launched its smart contract support with the Alonzo fork on September 12, 2021. Shortly after, a robust DeFi ecosystem began to develop, including the first Cardano DEXs: MuesliSwap, SundaeSwap, and MinSwap, all launching between late 2021 and early 2022. Initially, these DEXs faced challenges adapting Ethereum's DeFi protocols to the eUTxO model, prompting them to develop unique solutions. Although Cardano's DeFi ecosystem is valued at over $200 million, most scientific literature around DeFi focuses on Ethereum. Ethereum's account-based model and Cardano's eUTxO model exhibit significant differences, and we believe more focus should be given to DeFi within the eUTxO model. Therefore, this will be the primary focus of this work.

Structure of this work

In Chapter 2, we will begin by introducing key definitions that are essential to understanding this work's concepts, though these may already be familiar to those well-versed in blockchain-related fields.

In Chapter 3, we will analyze how different blockchains track users' balances, exploring each blockchain's accounting model. Specifically, we will define and discuss Bitcoin's UTxO model, Ethereum's account-based model, and Cardano's eUTxO model. We will also examine the concurrency issue and compare the eUTxO and account-based models.

In Chapter 4, we will formally define decentralized finance applications, focusing particularly on two DEX paradigms: Automated Market Makers and Limit Order Books, while discussing their properties.

In Chapter 5, we will explore the concept of Miner Extractable Value, which refers to the additional profits that rational block producers can earn by manipulating transaction inclusion and ordering.

In Chapter 6, we will apply the previously defined concepts to analyze three major Cardano DeFi projects, identifying a security vulnerability within each.

Finally, in Chapter 7, we will present our conclusions from the analysis of Cardano's DeFi ecosystem and highlight open challenges related to DeFi in the eUTxO model. We will also offer our perspective on the future direction of research in this area.

CHAPTER 2

PRELIMINARIES

2.1 Cryptography

Below, we will briefly define and discuss some cryptographic primitives that are used in all blockchain protocols. We will treat these primitives as black boxes, using them without delving into their theoretical definitions and analyses, as these aspects are orthogonal to our work.

Definition 2.1 (Hash function, [8])**.** A hash function (with output length ℓ) is a pair of probabilistic polynomial-time algorithms (Gen, H) satisfying the following:

- Gen is a probabilistic algorithm which takes as input a security parameter 1^n and outputs a key s. We assume that 1^n is implicit in s.
- H takes as input a key s and a string $x \in \{0,1\}^*$ and outputs a string $H_s(x) \in \{0,1\}^{\ell(n)}$, where n is the value of the security parameter implicit in s.

Definition 2.2 (Collision resistant Hash function, [8])**.** A hash function $\Pi = (\text{Gen}, H)$ is *collision resistant* if for all probabilistic polynomial-time adversaries A, there exists a negligible function negl such that

$$\Pr[\text{Hash-coll}_{A,\Pi}(n) = 1] \leq \text{negl}(n).$$

All the hash function we are interested in this work exhibit the collision resistance property.

Definition 2.3 (Signature Scheme, [8])**.** A (digital) signature scheme consists of three probabilistic polynomial-time algorithms $(\text{Gen}, \text{Sign}, \text{Vrfy})$ such that:

1. The key-generation algorithm Gen takes as input a security parameter 1^n and outputs a pair of keys (pk, sk). These are called the public key and the private key, respectively. We assume that pk and sk each have length at least n, and that n can be determined from pk or sk.
2. The signing algorithm Sign takes as input a private key sk and a message m from some message space (which may depend on pk). It outputs a signature σ, and we write this as $\sigma \leftarrow \text{Sign}_{\text{sk}}(m)$.

3. The deterministic verification algorithm Vrfy takes as input a public key pk, a message m, and a signature σ. It outputs a bit b, where $b = 1$ means valid and $b = 0$ means invalid. We write this as $b := \text{Vrfy}_{\text{pk}}(m, \sigma)$.

It is required that, except with negligible probability over (pk, sk) output by $\text{Gen}(1^n)$, it holds that $\text{Vrfy}_{\text{pk}}(m, \text{Sign}_{\text{sk}}(m)) = 1$ for every (legal) message m.

If there is a function ℓ such that for every (pk, sk) output by $\text{Gen}(1^n)$ the message space is $\{0,1\}^{\ell(n)}$, then we say that $(\text{Gen}, \text{Sign}, \text{Vrfy})$ is a signature scheme for messages of length $\ell(n)$.

We call σ a valid signature on a message m (with respect to pk and sk) if $\text{Vrfy}_{\text{pk}}(m, \sigma) = 1$.

By using a digital signature scheme, we can establish a notion of identity in a permissionless ad-hoc network such as a blockchain. Each entity in this network generates a public key and uses a series of hashes on it to create an address. Transactions originating from a particular address are signed with the user's private key. Henceforth, we will assume that this transaction validation process is performed each time. Valid transactions are accepted by the consensus mechanism participants and submitted for consensus, while invalid transactions are discarded.

2.2 Game theory

In game theory, strategic agents interact with one another, aiming to maximize their individual *payoff* or *utility*. The rules or circumstances that govern this strategic interaction can be collectively described as a game. Below, we will provide some basic game-theoretic definitions as presented in a number of introductory textbooks on game theory, such as [9]. We will also discuss some fundamental results of the profit-maximization mindset in blockchain, specifically how strategic behavior shapes the design of distributed mechanisms.

Definition 2.4 (Game, [9])**.** A *game* is defined as a tuple $G = (N, S, u)$, where:

- $N = \{1, 2, \ldots, n\}$ is the set of players.
- $S = S_1 \times S_2 \times \cdots \times S_n$ is the strategy space, where S_i is the set of *strategies* available to player i.
- $u = (u_1, u_2, \ldots, u_n)$ is the payoff or utility function, where $u_i : S \to \mathbb{R}$ gives the payoff to player i for each strategy profile $s = (s_1, s_2, \ldots, s_n) \in S$.

Definition 2.5 (Utility Function, [9])**.** A *utility* or *payoff* function $u_i : S \to \mathbb{R}$ is a function that assigns a real number (utility) to each strategy profile $s \in S$, representing the payoff or satisfaction player i receives when the strategy profile $s = (s_1, s_2, \ldots, s_n)$ is played.

Definition 2.6 (Expected Utility,[9])**.** For a set of outcomes $x_1, x_2, \ldots, x_n$ with corresponding probabilities $p_1, p_2, \ldots, p_n$, the expected utility for a player j is given by:

$$E[u_j] = \sum_{i=1}^{n} p_i u_j(x_i)$$

.

In game theory, each player is considered *rational*, i.e. they consistently choose strategies that maximize their utility, given their beliefs about the strategies of other players. Rationality implies that each player will select a strategy that provides the highest expected payoff. This assumption leads naturally to the next definition.

Definition 2.7 (Strictly Dominated Strategies, [9])**.** A strategy $s_i \in S_i$ for player i is *strictly dominated* if there exists another strategy $s'_i \in S_i$ such that for every possible combination of the other players' strategies $s_{-i} \in S_{-i}$, the following holds:

$$u_i(s'_i, s_{-i}) > u_i(s_i, s_{-i}) \quad \text{for all } s_{-i} \in S_{-i}.$$

This means that no matter what the other players do, player i will always receive a higher payoff by choosing s'_i over s_i.

Player rationality implies that a profit maximizing player will never choose a strictly dominated strategy.

2.2.1 Game theory in Blockchain; Classical results

In classical distributed computing literature, most models were constructed to account for participants that might go inactive (crash fault tolerance)[10] or be corrupted by an adversary (Byzantine fault tolerance)[11] during the execution of protocols. This focus stemmed from the fact that BFT mechanisms were originally designed to operate among a fixed number of known participants, such as a group of servers owned by a single company. Therefore, the threat model was limited to an adversary crashing or corrupting up to a fixed number of honest parties. The remaining honest parties were assumed to behave as the protocol intended. This made sense at the time because the honest parties were servers, usually geographically distributed across the planet, all owned by a single company.

With the rise of Bitcoin and permissionless consensus mechanisms, it became possible for parties to connect ad hoc to a network and participate in its consensus. Of course, numerous papers have analyzed Nakamoto-style consensus under the same assumptions as BFT, including the Bitcoin Backbone[3], a seminal paper by Garay, Kiayias, and Leonardos. These works were necessary for ensuring the security of Bitcoin and other Nakamoto-style protocols, but one wonders if they are sufficient. Specifically, the honest-adversary security model may fail to capture a third type of actor in a permissionless consensus protocol: the profit maximizer. This actor is neither honest, i.e., following the consensus protocol to the letter, nor an adversary who does everything in their power to harm the consensus. They are simply rational and utilize the consensus mechanism to maximize their utility, bending the rules slightly whenever it benefits them.

The first work to introduce the concept of *rationality* in blockchain protocols is [12]. In this work, Eyal et al. introduced the *selfish mining* strategy for block producers (BPs). Selfish mining involves a BP creating a block, B_1, and not immediately publishing it. Instead, they keep it private and work to extend their chain. When a new block, B_2, is published by a third party, they then publish their private block in an attempt to orphan B_2.

Definition 2.8 (Selfish Mining, [13])**.** The *Selfish-Mine* strategy can be summarized as follows:

- **Which Block:** Oldest block m.
- **Private Chain Length:** $Privatem$.
- **Publish Block** B**?:**
  - If $Height(B) = H$, then **yes**.
  - If $Racingm_H$ is true, and $Privatem = H + 1$, then **yes**.
  - Otherwise, **no**.

$Racingm_i$ is a boolean variable that is true if there are two blocks of height i, one produced by the miner m and the other by another miner.

Eyal et al. proved that:

$$u_i(s_{\text{selfish mining}}, s_{-i}) > u_i(s_{\text{honest}}, s_{-i}), \forall i \in N$$

Following the shellfish mining strategy, rather than being honest, strictly increases a BP's rewards, thus strictly dominating honest behavior. Additionally, the greater their hashing power as a percentage of the overall network's hashing power, the larger their reward. Consequently, BPs are incentivized to form increasingly large coalitions to execute the selfish mining strategy. Large miner coalitions pose a serious decentralization risk for Bitcoin and can lead to security breaches or forks[1].

2.3 The Cardano blockchain

Below, we will analyze several key concepts of the Cardano blockchain. We will break down our analysis into three separate layers, namely the *Network Layer*, the *Consensus Layer*, and the *Application Layer*. Each layer operates on top of the others.

2.3.1 Network Layer

Currently, the Cardano network operates under the "dynamic Peer-to-Peer" model, as described in the IOG[2] documentation [14] [15].In this model, there are two distinct types of nodes:

- **Block-producing nodes**: These nodes are responsible for participating in Cardano's consensus mechanism. Block producers run block-producing nodes that require incoming connections to receive block information and outgoing connections to propagate the blocks they generate. We will explore the responsibilities of BPs in more detail later in this chapter, particularly as they relate to their role in the consensus mechanism.
- **Relay nodes**: These nodes are responsible for communicating with other relays in the network and broadcasting blocks from block-producing nodes. There is a

[1] At the time of writing, the two largest mining pools control more than 51% of the total hashing power in Bitcoin. You can check the hashing power allocated across different entities in Bitcoin in real-time here: https://hashrateindex.com/hashrate/pools

[2] Input Output Global (IOG) was the company responsible for developing Cardano until the Chang hard-fork, when the responsibility for development and governance of the Cardano blockchain shifted to the community, utilizing on-chain governance tools for decision-making

subcategory of relay nodes, known as *trusted* relays. A trusted relay is responsible for providing accurate on-chain data for decentralized applications (DApps), i.e., protocols that utilize the Cardano blockchain for data storage and operate in a decentralized manner. Each DApp has a different set of trusted relays.

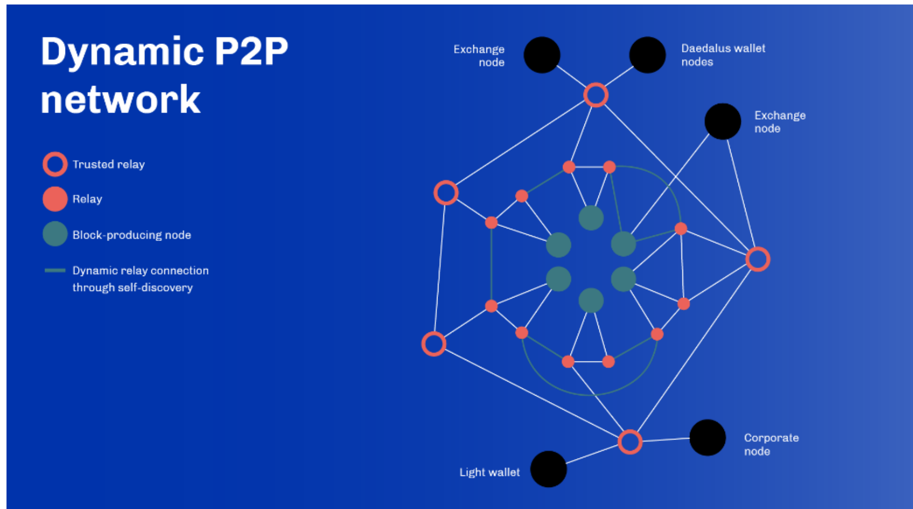

Figure 2.1: A visualization of the Cardano network topology, as given in [15]

Relay nodes act as an additional defense layer, securing a block-producing node from being directly accessible to adversarial entities on the network. The protocol requires that each Block Producer operates one block-producing node and two relay nodes. Due to the anonymity of the network, a Block Producer may control more relay nodes.

Essentially, relay nodes are utilized by Block Producers in a gossip protocol [16]. A gossip protocol is a method for information propagation in distributed and ad-hoc networks like Cardano.

2.3.2 Consensus layer

At the time of writing, Cardano uses Ouroboros Praos [17], the second iteration of the Ouroboros [18] protocol family, designed by Kiayias et al. Ouroboros is a longest-chain (Nakamoto-style) Proof-of-Stake consensus protocol. We will briefly discuss and analyze some primitives of the Ouroboros protocol that are relevant to our work and are, at this point, widely known, while treating all other aspects of Ouroboros as a black box.

Cardano employs the Ouroboros consensus mechanism to maintain a robust public transaction ledger. The entities that use this transaction ledger to maintain and exchange value are called *users*. The entities responsible for maintaining and updating the transaction ledger are called *miners* or *Block Producers* (**BPs**).

Transactions are submitted by users to BPs for inclusion in the ledger. BPs check if these transactions are valid and subsequently *order* these transactions. The execution

of these transactions happens serially as they are ordered. This ordering is encoded as a *block*. A block consists of its header and its body. The body consists of the ordered transactions and other consensus-specific information (such as a nonce and the timestamp). The header is a hash of the body, thus encoding the block in an *immutable* manner. In turn, blocks are ordered by the consensus mechanism, forming a chain of blocks or a *blockchain*. The time window in which a block is created and validated through consensus is named a *round* of the protocol.

Note that users do not participate in the consensus and, thus, are not able to submit transactions themselves. Instead, they must submit their transactions through a BP. Due to the distributed nature of the blockchain, each BP has knowledge of different submitted transactions waiting for block inclusion. This is known as a BP's *local knowledge* or *mempool* (short for memory pool).

Finally, BPs are incentivized to maintain the ledger by the ledger itself. This is done by BPs receiving fees for including each transaction in a block. We note that, compared to other blockchains like Ethereum, in Cardano the users do *not* decide the amount of this fee. Instead, the fee is computed by a predefined rule of the protocol and is proportional to each transaction's complexity and memory requirements.

2.3.3 Application Layer

The application layer consists of decentralized applications (DApps) that run on top of the blockchain. These DApps function by deploying smart contracts on the blockchain. A smart contract is a self-executing program with predefined rules that, when certain conditions are met, automatically executes the encoded logic, such as triggering a transaction or updating a state.

On the Cardano blockchain, smart contracts are written using functional programming languages, making them resemble functions that produce deterministic outputs for specific inputs. In this work, we assume that smart contracts behave as intended by their creators and do not delve into bugs that may arise from programming errors. One can think of Smart contracts[3] imbuing programmability on the blockchain.

In [21], Warner et al identify four properties that smart contracts need to have in order for DeFi applications to run on top of them. They must:

1. Be expressive enough to encode protocol rules.
2. Allow conditional execution and bounded iteration.
3. Be able to communicate with one-another, via message-calls, within the same execution context (typically a transaction).
4. support atomicity, i.e., a transaction either succeeds fully or fails entirely.

Of course, not all data used by DApps comes from smart contract execution. Many DApps need to track on-chain information about their users, such as asset ownership and account balances. To achieve this, as explained earlier, each DApp relies on a set

[3]Interestingly, the concept of smart contracts predates blockchain technology, originating in 1997[19], [20].

of trusted relay nodes. These relay nodes are responsible for providing accurate and up-to-date data to their respective DApps.

CHAPTER 3

LEDGER MODELS

Much like traditional banking, in a distributed ledger, each user possesses accounts with which they transact with one another. Due to the financial nature of the application, we should be careful so users cannot double spend tokens, i.e., the total balance on their accounts should not exceed the tokens that they can spend in future transactions.

There are several ways to represent the abstract notion of *account balance*, and each one is specifically tied to the underlying blockchain on top of which the distributed ledger is maintained. The account balance abstraction is known as *ledger model* [22]. The two most dominant paradigms are Bitcoin's *UTxO* model and Ethereum's *account-based* model.

Ethereum turned away from Bitcoin's UTxO paradigm because, at that time, it was believed that the UTxO model could not support additional programmability. Hence, the account-based model was developed with smart contract support in mind. Later, Cardano's team introduced a generalization of the UTxO model, named the *extended UTxO model*, which was indeed capable of supporting smart contracts. In the rest of this chapter, we will take a deep dive into how transactions are represented in each ledger model.

3.1 The account-based model

The account-based model can be thought of as operating similarly to a digital banking system. In this model, the decentralized ledger maintains a database of each user's outstanding account balance, i.e., the amount of token X that each user possesses and has not yet spent. A user's balance for each token she possesses is encoded in her account state. The mapping between account addresses and account states is known as the *world* or *global state* of the blockchain.

Definition 3.1 (World state, [23])**.** The **World State** in Ethereum can be represented as a mapping:

$$\text{WorldState} : \text{Address} \rightarrow \text{AccountState}$$

Each account in Ethereum is represented by an **account state**, which is a tuple:

$$\text{AccountState} = (\text{nonce}, \text{balance}, \text{storageRoot}, \text{codeHash})$$

- **nonce**: A scalar value that tracks the number of transactions sent from the account (for externally owned accounts) or the number of contract creations (for contract accounts).
- **balance**: A scalar value equal to the amount of ETH (Ethereum's native token) owned by the account.
- **storageRoot**: The root of the Merkle Patricia Trie that encodes the storage content of the account.
- **codeHash**: The hash of the EVM (Ethereum Virtual Machine) bytecode associated with the account. For externally owned accounts, this field is empty.

We will now give a simple example to illustrate how the account-based model operates. In this scenario, we assume that each entity identifies herself on the blockchain by leveraging an existing *public key infrastructure* (**PKI**), but we abstract away this fact for simplicity. Let's suppose that both Alice and Bob have a balance of 100 tokens of X in their accounts and Alice wants to send Bob 30 tokens. She also needs to pay a transaction fee of 2 tokens. Let's see how this transaction will be processed in this model.

Alice submits her transaction to a BP. The BP will query her local view of the blockchain and verify that Alice indeed has enough balance in her account to execute this transaction. After Alice's transaction has been included in the blockchain, Alice's and Bob's account balances will be adjusted accordingly.

Notice that to describe the above-mentioned example, the notion of coin or token identity was not needed. In the account-based model, each user account is simply a list of their owned assets, and tokens are indistinguishable from one another.

3.1.1 Ethereum Transactions

The account-based model is formally described by Gavin Wood, an Ethereum co-founder, in Ethereum's Yellow Paper [23]. In this construction, Ethereum can be viewed as a *transaction-based state machine*.

Definition 3.2 ([24])**.** A *state machine* is defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (the alphabet).
- $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function, mapping a state and input symbol to a new state.

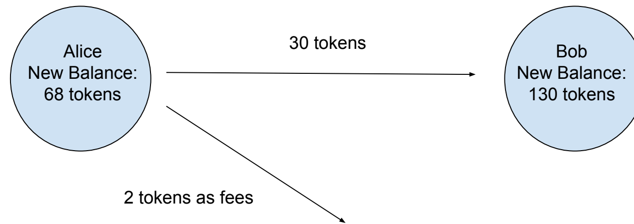

Figure 3.1: A transaction in the account-based model. The edges represent transactions, and the nodes represent accounts.

- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of accepting (or final) states.

For more information about state machines, see [24] by Hopcroft, Motwani and Ullman.

Definition 3.3 (Ethereum canonical version,[23],simplified)**.** To adapt the state machine definition to Ethereum's notation:

- **State (σ)**: The global state of Ethereum at time t, denoted σ_t. It encodes information such as account balances, contract storage, etc.
- **Initial State (σ_0)**: This in an arbitrary initial state from which all BPs start from. Specifically in Ethereum, it is its Genesis block.
- **Transactions (T)**: Operations that trigger state transitions. Each transaction represents valid arcs between states.
- **Ethereum state transition function (Υ)**: Defines how the global state σ_t is transformed into a new state σ_{t+1} based on the execution of a transaction T:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

- **Blocks (B)**: Transactions are grouped into blocks by a resource-intensive method like a PoW or PoS. Formally:

$$B \equiv (\ldots, (T_0, T_1, \ldots), \ldots)$$

- **Block-level state transition function**(Π): Determines how the state is updated based on the transactions within a block:

$$\sigma_{t+1} \equiv \Pi(\sigma_t, B) \equiv \Upsilon(B, \Upsilon(\Upsilon(\sigma_t, T_0), T_1), \ldots)$$

There are several things to be unpacked in this definition, in particular, what happens in the BP's local view, and what happens in the blockchain's global view.

Blocks are produced by BPs and contain not-yet-executed transactions that were part of their local mempool at the time of block creation. Hence, to produce a valid block, a BP needs to apply the state transition function until her local mempool is empty or the block size limit has been reached. Notice that a BP has complete control over what transactions are included in the block and in what order, as long as all transactions are valid. We will then see that this ability, coupled with the fact that BPs in our model are rational, creates additional complexities in DeFi.

Let us now shed some light on how BPs are synchronized to update the global state in each epoch. When a valid block is produced, it is propagated in the network. Then, BPs run Ethereum's BFT-style consensus mechanism. The way consensus is achieved is orthogonal to our work, and we won't go into much detail here. For more information about it, see [25] by Vitalik and Griffith. After all fork resolution rules have been applied and consensus has been reached, each BP accepts the same block. With it, she applies the block transition function to her previous local state, arriving at a new one. Observe that all BPs begin from the same initial state (the Genesis block), and in each round, apply the block transition function with the same block. Hence, they are synchronized once again to achieve a single global state.

Definition 3.4 (Account-based transaction, [26])**.** An account-based transaction is defined as:

Acctx $:=$ (*sender* : Address, *receiver* : Address, *value* : Value, *forge* : Value, *fee* : Value)

Where:

- The sender field indicates the sender's address.
- The receiver field indicates the receiver's address.
- The value field indicates the transaction's value, denominated in native tokens.
- The forge field indicates the total amount of tokens created in this transaction. It is only relevant for transactions that mint new tokens, often called *coinbase transactions*. Coinbase transactions are the only type of transactions to which the conservation law does not apply. A blockchain's minting policy is outside the scope of this work and thus is only included here for completeness.
- The fee field indicates the amount of tokens transferred to the BP of the block in which this transaction is included.

3.2 The UTxO model

We now turn to the Unspent Transaction Outputs (**UTxO**) ledger model, first encountered in Bitcoin. As the name suggests, UTxO blockchains treat each individual account balance as the sum of the value of the *unspent* transactions belonging to that account.

Let’s examine how transactions in the UTxO setting look. Again, because we are only interested in the way transactions are created and recorded in the ledger, we will abstract away many of the technical details. Value (denominated in tokens) is grouped together and encoded by cryptographic hashing. These coalition resistant hash encodings of value are known simply as *Unspent Transaction Outputs* (**UTxOs**). Each UTxO transaction has inputs and outputs. Its inputs are the UTxO hashes of previous transactions which are not yet spent. Its outputs are also UTxO hashes, such that the total value of the input transactions has to be equal to the total value of the output transactions. This property is known as the *conservation law*. Summing up, a UTxO transaction consumes UTxOs and creates new ones, while maintaining the total value.

Definition 3.5 (Conservation Law, [27], adjusted)**.** If υ represents the value of a transaction then for every transaction is must hold that:

$$\sum_{i \in \text{UTxO input set}} \upsilon_i = \sum_{j \in \text{UTxO output list}} \upsilon_j$$

Much like the account-based model is akin to digital banking, accounting in the UTxO setting closely resembles cash transactions. Hence, Bitcoin transactions operate like cash transactions, except for some improvements, as elegantly captured by Zahnentferner in [26]:

- **Arbitrary Denominations**: Cash has fixed denominations (e.g., $1, $5, $10), while Bitcoin allows arbitrary amounts encoded in UTxOs.
- **Transaction Process**: In cash transactions, physical coins and notes are passed intact. In Bitcoin, UTxOs are destroyed when spent and new ones of equal value are created.
- **Exact Payment**: With cash, if the exact amount isn't available, the buyer overpays and waits for change. In Bitcoin, the buyer can send the exact payment and receive the change in the same transaction.

Again, we will illustrate the way that the UTxO model operates by revisiting our previous example. Let’s suppose that Alice has a balance of 100 tokens, sends Bob 30 tokens, and also pays 2 as fees. In our cash analogy, Alice would go to a bank, exchanging one of her two $50 bills for change. Then she would give $30 to Bob, $2 for fees, and keep the change ($18).

To spend tokens in this setting, Alice needs to reference her previous UTxOs as input in her new transaction. Let’s assume that Alice owns two UTxOs from previous transactions, each one having a value of 50 tokens. Alice creates a transaction with one of the 50-token UTxOs as input. This transaction will create three UTxO outputs: one

belonging to Bob with a value of 30 tokens, one belonging to the BP of Alice's block with a value of 2 tokens, and one belonging to Alice with a value of 18 tokens. Each of these output transactions is now owned by their corresponding entities and can be used as input to future transactions.

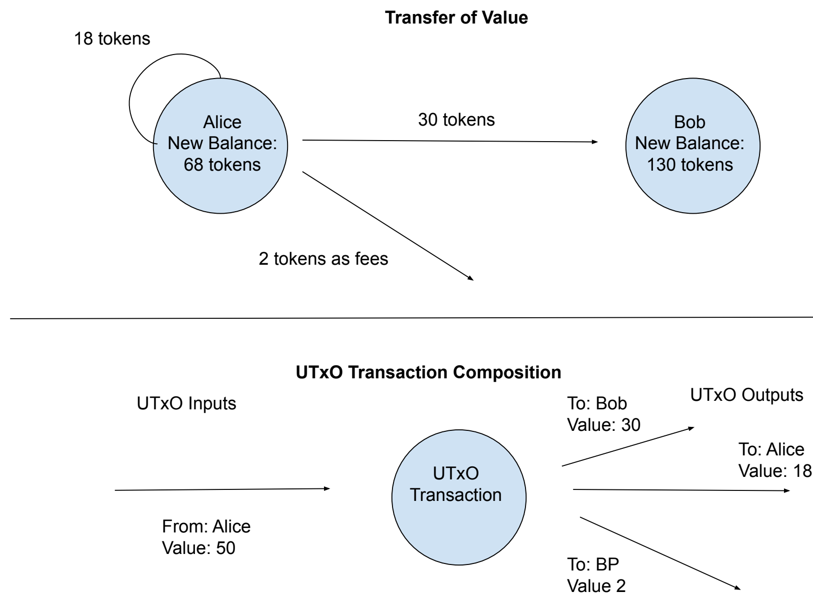

Figure 3.2: A transaction in the UTxO model.

When Alice submits her transaction to the BPs for inclusion in the blockchain, the BPs need to validate that the Conservation Law is upheld. They also need to validate that Alice has not already spent her UTxO inputs. To do so, BPs hold a list of all unspent UTxOs, which is known as the *UTxO set*. In each round, when new blocks are created and propagated throughout the network, each BP participates in the consensus mechanism to determine which block to accept. From the agreement property, each honest BP will accept the same block. Thus, after the round, all honest BPs' UTxO sets will be identical.

Overall, a UTxO transaction consists of a set of inputs and a list of outputs, where outputs represent a specific UTxO that can be spent as an input of future transactions. Each output can be spent by exactly one input. Thus, cycles are not allowed in these connections, so a collection of transactions that spend from one another can be viewed as a *directed acyclic graph* (**DAG**) [27]. In this DAG, a transaction with m inputs and n outputs is represented by a node with m incoming edges and n outgoing edges. Additionally, the conservation law must be upheld in each internal node.

3.2.1 UTxO Transactions

Definition 3.6 (UTxO transaction output,[26], adjusted)**.** A UTxO transaction output is in the form of:

$$\text{Output} \coloneqq (\textit{address} : \text{Address}, \textit{value} : \text{Value})$$

where the Address field corresponds to the UTxO's owner, and the Value field indicates the UTxO's value denominated in native tokens.

Definition 3.7 (UTxO transaction input,[26], adjusted)**.** A UTxO transaction input is in the form of:

$$\text{Input} \coloneqq (\textit{id} : \text{Id}, \textit{index} : \text{Int})$$

where the id field references a previous unspent transaction that this input refers to, and the index field indicates which of the referred transaction's ordered outputs should be spent in this transaction.

Definition 3.8 (UTxO transaction,[26], adjusted)**.** A UTxO-based transaction is defined as:

$$\text{UTxOtx} \coloneqq (\textit{inputs} : \text{Set[Input]}, \textit{outputs} : \text{List[Output]}, \textit{forge} : \text{Value}, \textit{fee} : \text{Value})$$

Where:

- Inputs are elements of the UTxO set.
- Outputs are ordered in a list so that they can be referenced by future transactions.
- The forge field indicates the total amount of tokens created in this transaction. It is only relevant for transactions that mint new tokens, often called *coinbase transactions*. Coinbase transactions are the only type of transactions the conservation law does not apply to. A blockchain's minting policy is outside the scope of this work and thus is only included here for completeness.
- The fee field indicates the amount of tokens transferred to the BP of the block that this transaction is included in.

3.2.2 Bitcoin Transactions

The UTxO ledger model definitions we presented in this section help establish some intuition about the inner workings of the UTxO model, but they don't exactly reflect the reality of Bitcoin. We will amend this though, following Zahnentferner's definitions on [28]. To do so, we first have to introduce two new primitives, namely *redeemers* and *validators.*

The UTxO ledger model definitions we presented in this section help establish some intuition about the inner workings of the UTxO model, but they don't exactly reflect the reality of Bitcoin. We will address this by following Zahnentferner's definitions in [28]. To do so, we first need to introduce two new primitives, namely *redeemers* and *validators*.

A validator is a script associated with each transaction output in Bitcoin. Unlike the abstracted UTxO model, where the owner's address is encoded in the UTxO, the validator script defines the conditions that must be met for a UTxO to be spent in future transactions. When a transaction is created, each output includes this validator script, which is an algorithm that determines how a UTxO should be spent. Usually, the validator checks if the provided signature matches the corresponding private key, although its logic can be more complex.

A redeemer is a script provided by a transaction input that interacts with the corresponding validator script of the output it aims to spend. When a transaction input attempts to spend a UTxO, its redeemer script supplies the necessary data or proof required by the validator script to authorize the spending. If the redeemer script successfully meets the criteria defined by the validator script, the input is considered valid, and the transaction can proceed. If it fails, the transaction is invalid, and the output cannot be spent.

Together, the validator and redeemer scripts can be thought of as a lock-and-key system, where the validator script locks the output with specific conditions, and the redeemer script unlocks it by providing the required proof or data.

Definition 3.9 (Bitcoin transaction output, [28], adjusted)**.** A Bitcoin transaction output is in the form of:

$$\text{Output} \coloneqq (\mathit{validator} : \text{Script}, \mathit{value} : \text{Value})$$

where the validator field corresponds to a cryptographic hash of the validator script (that is kept off-chain), and the value field indicates the UTxO's value denominated in native tokens. Notice that there is no mention of the owner's address in this iteration of the UTxO model. The proof of ownership is left to the validator script, which is accomplished using cryptography.

Definition 3.10 (Bitcoin transaction input, [28], adjusted)**.** A Bitcoin transaction input is in the form of:

$$\text{Input} \coloneqq (\mathit{id} : \text{Id}, \mathit{index} : \text{Int}, \mathit{redeemer} : \text{Script})$$

where the id field references a previous unspent transaction that this input refers to, and the index field indicates which of the referred transaction’s ordered outputs should be spent in this transaction. The redeemer field contains the appropriate script that will interact with the validator script and allow the referenced UTxO to be spent.

Definition 3.11 (Bitcoin transaction, [28], adjusted)**.** A Bitcoin transaction defined as:

$$\text{BUtxoTx} \coloneqq (\mathit{inputs} : \text{Set[Input]}, \mathit{outputs} : \text{List[Output]}, \mathit{forge} : \text{Value}, \mathit{fee} : \text{Value})$$

where each field is identical to the corresponding UTxO transaction definition given above.

3.3 The eUTxO model

Due to the UTxO model's limited expressiveness, it was believed that it could not support complex transaction logic, such as escrow deposits or loans. However, Chepurnoy et al. proved in [29] that a UTxO blockchain is, in fact, Turing-complete [30], and therefore capable of supporting smart contracts.

Following up, Chakravarty et al. introduced the *extended Unspent Transaction Outputs* (**eUTxO**) model in [22]. The eUTxO model is an expansion of the UTxO model, inheriting all of its characteristics and properties. Specifically, as in Bitcoin, each transaction references previous UTxOs as inputs and produces UTxOs as outputs, which can be referenced by future transactions. Additionally, the eUTxO model employs the *redeemer* and *validator* primitives.

Additionally, they augmented the standard UTxO model with these three novel primitives:

- **Datum**: An additional piece of data carried by all transactions, passed on as an additional input to the validator script. This enables a smart contract to store some notion of state without altering the validator code.

- **Context**: Additional information about the transaction that is being validated. It enables the validator to enforce much stronger conditions than is possible with a bare UTxO model. In particular, it can inspect the outputs of the current transaction. The information provided by Context is entirely determined by the transaction itself.

- **Validity Interval**: The time interval in which a transaction can be processed. It is measured in *ticks*, i.e., a monotonically increasing unit of progress in the ledger system. It corresponds to the block number or block height.

To illustrate the eUTxO model, we will reiterate the example given by Chakravarty et al. in [28], a multisignature wallet. Specifically, we wish to create a smart contract that locks value υ and has a set of m signatures hard-coded into it. In order for υ to be spent, at least n signatures need to be collected $(m > n)$. We will construct a state machine, use the datum δ to save the state, and use the validator-redeemer interaction to update the state as needed. See Figure 3.3 for the transition diagram of this state machine.

Name	Symbol	Meaning
Transaction	tx	A transaction on the decentralized ledger
Redeemer	ρ	The redeemer script for one of the transaction's UTxO inputs
Value	υ	The value of a transaction denominated in native tokens
Datum	δ	A piece of data containing contract-specific data.
Validator	$V(\cdot)$	The validator script. A transaction is valid if V$(\upsilon, \delta, \rho, tx) = 1$

Table 3.1: List of symbols in eUTxO

- A validator function V along with the datum δ is used to lock υ.

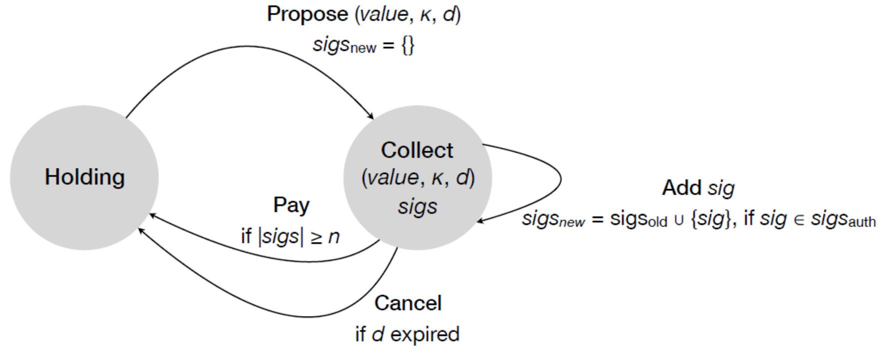

Figure 3.3: Transition diagram for the multi-signature state machine; edges labelled with input from redeemer and transition constraints. As given in [22]

- The datum δ stores the machine state, which can take the form of:
  - `Holding`, when only the locked value υ is held.
  - `Collecting`((υ, κ, d), `sigs`), when collecting signatures `sigs` for a payment of υ to κ by the deadline d.
- The initial output for the contract is $(V, \upsilon, \texttt{Holding})$.
- The validator V implements the state transition diagram from Figure 3.3 by:
  - Using the redeemer ρ of the spending input to determine the required state transition.
  - Validating that the spending transaction tx is a valid representation of the newly reached machine state. This requires:
    * Ensuring tx keeps υ locked by V.
    * Verifying that the state in the datum δ' is the successor state of δ according to the transition diagram.
- The redeemer ρ (state machine input) can take one of four forms:
  - `Propose(`υ`, `κ`, d)`: Proposes a payment of υ to κ by the deadline d.
  - `Add(sig)`: Adds a signature `sig` to a payment.
  - `Cancel`: Cancels a proposal after its deadline has expired.
  - `Pay`: Makes a payment once all required signatures have been collected.

3.3.1 eUTxO Transactions

Bellow, we will state definitions for the eUTxO transactions, as given in [22].

Definition 3.12 ([22], Adjusted)**.** A eUTxO transaction output is in the form of:

$$\text{Output} := (\textit{value} : \text{Quantity}, \textit{addr} : \text{Address}, \textit{datumHash} : \text{DataHash})$$

where the value field indicates the value of the UTxO denominated in native currency, the address field is a hash of the UTxO's validator script and the datumHash field is a hash of the transactions data.

Definition 3.13 ([22], Adjusted)**.** A eUTxO transaction input is in the form of:

$$\text{Input} = (\textit{outputRef} : \text{OutputRef}, \textit{validator} : \text{Script}, \textit{datum} : \text{Data}, \textit{redeemer} : \text{Data})$$

where:

- The outputRef field references a previous unspent transaction that this input refers to.
- The validator field indicates this transaction's validator script.
- The datum field indicates this transaction's datum data.
- The redeemer indicates field this transaction's redeemer script.

Definition 3.14 ([22], Adjusted)**.** A eUTxO transaction is defined as:

$$\text{eUTxOTx} = (\textit{inputs} : \text{Set[Input]}, \textit{outputs} : \text{List[Output]}, \textit{validityInterval} : \text{Interval[Tick]})$$

where the inputs field indicates the transaction's input set, the outputs field indicates this transaction's output set and the validityInterval field indicates the amount of time (measured in ticks) this transaction has to be validated in. We assume that there is some notion of a current tick for a given ledger.

3.3.2 Concurrency

Concurrency is an issue in the UTxO and eUTxO models. As we have seen during this chapter, a UTxO transaction references and consumes UTxOs to produce new ones. This means that if there are multiple transactions referencing the same UTxO, all but one of them will fail.

To unpack the meaning behind this statement, let's suppose that both Alice and Bob submit transactions to be included in the same block, tx_A and tx_B respectively. Also, both of these transactions reference the same UTxO, $UTxO_1$. Without loss of generality, we suppose that Alice's transaction is processed first. This results in tx_A consuming the UTxO to produce a new one, $UTxO_2$. Hence, $UTxO_1$ is removed from the UTxO set. When Bob's transaction is processed next, it would be referencing a UTxO not belonging to the UTxO set, and thus will fail. Notice that things don't get better for Bob in subsequent turns; the UTxO his transaction tries to reference will never be back in the UTxO set.

Concurrency is not a major problem in Bitcoin because the only type of transaction is a transfer between accounts. Hence, if Alice wants to transfer BTC to both Bob and Charlie (Tx_B and Tx_C respectively), she either has to reference different UTxOs for each transaction or execute Tx_B first and then reference the resulting UTxO in Tx_C.

In both cases, she simply needs to plan her transactions accordingly.

This is not the case for eUTxO blockchains that support smart contracts, such as Cardano. Due to the smart contract capabilities of Cardano, several DeFi projects operate on it. These DeFi projects deploy smart contracts on Cardano to implement various DeFi applications. Because of the eUTxO model, users can interact with those smart contracts by referencing them as UTxOs in their transactions. However, due to the popularity of these smart contracts, there are usually several different users trying to transact with them at any given moment, often within the timespan of a single block's creation. As we explained above, only one of these transactions can be valid before the smart contract's UTxO is altered by it. Thus, only one of these transactions can be included in a single block. Concurrency creates a bottleneck [1] for the scalability of an eUTxO blockchain and needs to be addressed by new primitives, resulting in the off-chain processing of these transactions. We will analyze how concurrency affects Cardano's DeFi in later chapters.

3.4 eUTxO and account-based comparison

Brünjes and Gabbay provide an in-depth comparison between the eUTxO and account-based models in [31]. In their work, they formally highlight how each model operates on a technical level and compare their effectiveness concerning the blockchain's concurrent and distributed nature.

Ethereum's account-based model naturally aligns with an imperative programming style, which is by far the most popular style among active developers today. Additionally, in the account-based model, each account is simply a list of asset ownership. This abstraction feels natural and quite intuitive, which explains why most of the blockchain industry works on Ethereum[2] or an other EVM framework. In Ethereum, a smart contract is a program that modifies a global state by mapping global variables, i.e., accounts, to values. Therefore, Ethereum's smart contracts need to explicitly know the global state to operate correctly, which is not trivial considering the blockchain's highly concurrent nature. This has led to many security breaches, including the DAO hack, which cost Ethereum's ecosystem $70 million.

On the other hand, Cardano's eUTxO is best implemented using a functional programming approach. In this model, a smart contract is a function that takes a UTxO-list as its input and returns a UTxO-list as its output, with no other dependencies. Brünjes and Gabbay proved that a UTxO transaction is either valid (and thus included in a block) or invalid. Most importantly, the validity and outputs of a UTxO transaction can be determined by considering only its inputs (i.e., the UTxO set). Hence, if a transaction is submitted for inclusion in the blockchain, at worst, other entities might prevent its inclusion. However, if the transaction is included in the blockchain, its owner obtains the originally intended result. This concept is known as *transaction determinism*, and is pivotal for DeFi-specific applications. We will highlight its effects in the upcoming chapters.

[1] This became apparent to users during the launch of SundaeSwap, Cardano's biggest DEX: https://cointelegraph.com/news/sundaeswap-launches-on-cardano-but-users-report-failing-transactions

[2] For a rundown of developer activity in open-source blockchain projects, see: `https://www.developerreport.com/`

Example

Finally, we will provide a simple example to illustrate the above-mentioned point, involving three parties: Alice, Bob, and Charlie. We will see how the same series of transactions are handled by the eUTxO and the account-based models.

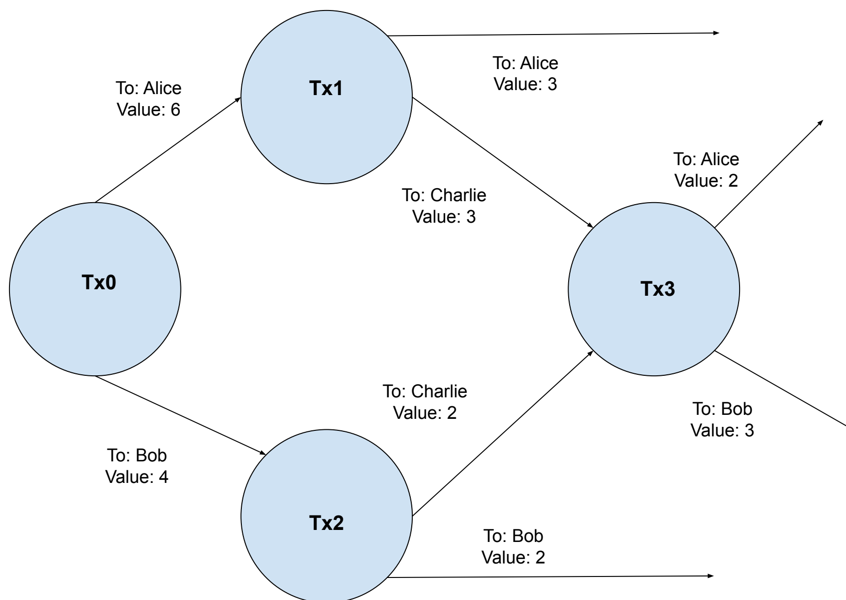

Figure 3.4: A series of transactions involving three parties: A, B and C. The incoming edges represent an outstanding account balance for A and B.

eUTxO Example

In the eUTxO model, each transaction references specific UTxOs as inputs, and these UTxOs are consumed to produce new UTxOs. Consider the following series of transactions involving Alice (A), Bob (B), and Charlie (C):

- **Tx0:** Initial transaction that creates two UTxOs:
  - 6 ADA sent to Alice (A).
  - 4 ADA sent to Bob (B).
- **Tx1:** Alice sends 3 ADA to Charlie (C) from the 6 ADA she received.
- **Tx2:** Bob sends 2 ADA to Charlie (C) from the 4 ADA he received.

- **Tx3:** Charlie sends 2 ADA back to Alice and 3 ADA to Bob from the 5 ADA he received in Tx1 and Tx2.

The sequence of transactions in the eUTxO model is as follows:

Transaction	Input (UTXO)	Output	Resulting Balances
Tx0	$O \to A$ $O \to B$	$6 \to A$ $4 \to B$	A: 6 ADA, B: 4 ADA, C: 0 ADA
Tx1	$A \to C$	$3 \to C$	A: 3 ADA, B: 4 ADA, C: 3 ADA
Tx2	$B \to C$	$2 \to C$	A: 3 ADA, B: 2 ADA, C: 5 ADA
Tx3	$C \to A$ $C \to B$	$2 \to A$ $3 \to B$	A: 5 ADA, B: 5 ADA, C: 0 ADA

Account-Based Model Example

In the account-based model, transactions update the balances of accounts directly. However, this model can lead to issues if transactions are processed in an arbitrary order. Consider the following sequence of transactions in two different orders:

First Order (All Transactions Valid):

- **Tx0:** Alice (A) receives 6 ETH, and Bob (B) receives 4 ETH.
- **Tx1:** Alice sends 3 ETH to Charlie (C).
- **Tx2:** Bob sends 2 ETH to Charlie (C).
- **Tx3:** Charlie sends 2 ETH back to Alice and 3 ETH to Bob.

The resulting balances after each transaction in the account-based model:

Transaction	A (ETH)	B (ETH)	C (ETH)
Tx0	6	4	0
Tx1	3	4	3
Tx2	3	2	5
Tx3	5	5	0

Second Order (Some Transactions Invalid):

- **Tx0:** Alice (A) receives 6 ETH, and Bob (B) receives 4 ETH.
- **Tx1:** Alice sends 3 ETH to Charlie (C).
- **Tx3:** Charlie sends 2 ETH to Alice and 3 ETH to Bob (invalid).
- **Tx2:** Bob sends 2 ETH to Charlie (C).

In this scenario, the transaction **Tx3** is processed before **Tx2**, which leads to an invalid transaction because Charlie (C) would not have enough ETH to send both 2 ETH to Alice and 3 ETH to Bob before receiving the 2 ETH from Bob.

The resulting balances in the account-based model when processed in the wrong order:

Transaction	A (ETH)	B (ETH)	C (ETH)
Tx0	6	4	0
Tx1	3	4	3
Tx3	5	5	(Invalid)
Tx2	3	2	5

Key takeaways

In the eUTxO model, the sequence of transaction processing is enforced by the UTxO input set of each transaction. Hence, it is not possible for **Tx3** to be included in the blockchain before **Tx2**, since **Tx3** references one of **Tx2**'s outputs. It will either be included after **Tx2** or it would be invalid. To rephrase this in graph-theoretic terminology, a valid UTxO transaction execution sequence is one that respects the topological ordering of the UTxO DAG.

In this account-based model example, the arbitrary ordering of transactions can lead to invalid transactions, which poses a risk to the integrity of the transaction process.

CHAPTER 4

DECENTRALIZED FINANCE

Unlike traditional financial systems, which rely on centralized institutions like banks, decentralized finance (**DeFi**) projects leverage blockchain smart contracts to enable users to access financial services without the need for intermediaries.

Perhaps the most popular DeFi application is decentralized exchanges (**DEXs**). A DEX project deploys smart contracts that facilitate trading between tokens on the blockchain. In [21], Warner et al. gave a brief rundown of the ideal properties of a DeFi application, namely:

1. **Non-custodial**: participants have full control over their funds at any point in time.
2. **Permissionless**: anyone can interact with financial services without being censored or blocked by a third party.
3. **Openly auditable**: anyone can audit the state of the system.
4. **Composable**: its financial services can be arbitrarily composed such that new financial products and services can be created.

There are two prominent DEX paradigms: *Automated Market Makers* (**AMMs**) and *Limit Order Books* (**LOBs**). In the remainder of this chapter, we will provide an overview of the DeFi infrastructure, define AMMs and LOBs, and compare their respective strengths and weaknesses.

We will first discuss LOBs. The concept is straightforward and closely resembles how traditional financial institutions (e.g., stock exchanges) operate. Specifically, each party that wishes to trade a certain asset pair (e.g., asset X for asset Y) declares their intent, the amount of the asset they wish to trade, and their asking price. For example, let's say Alice wants to sell 100 units of asset X for at least 80 units of asset Y. This is a limit order, i.e., an order that is executed only of a certain price limit is met. A collection of such limit orders is called a limit order book, or LOB for short. An on-chain LOB utilizes the blockchain to store and match buy orders and sell orders with one another. The entities responsible for matching buy orders to sell orders are called

market makers and are usually the BPs or some trusted third party. Note that in a LOB, an order will be executed only when it is matched. Hence, a trader might have to wait a while for their order to be executed.

On the other hand, as the name suggests, an Automated Market Maker tries to automate this process, offering near-instant transaction execution. The trade-off is that in an AMM, traders only have control over their intent and the amount of the traded asset, but not the price. Instead, the price is set by an on-chain mechanism, which is publicly verifiable by anyone querying the blockchain. So, returning to our previous example, instead of Alice being matched with Bob to trade, Alice deposits 100 units of token X into a smart contract to receive some amount of token Y, based on the current exchange rate at that time. This type of smart contract is called a *liquidity pool* because it offers the required liquidity for the trade to be made in exchange for a predetermined fee. It obtains said liquidity from other users called *Liquidity Providers* (**LPs**). We will explore AMMs and LOBs in much more detail later in this chapter.

4.1 Types of Security

DeFi is a relatively new and vibrant multidisciplinary area that encompasses primitives from many other fields such as distributed computing, algorithmic game theory, computational finance, and cryptography. Each area intersects with the others in fine points, and to achieve overall security, a protocol needs to be safe across all the aforementioned areas.

Before providing concrete definitions of DeFi, it is prudent to discuss different DeFi security notions. DeFi has grown rapidly in recent years, and there has been no shortage of incidents in which users' funds were stolen. These incidents gave rise to two types of security models: *technical security* and *economic security*.

Among the first and most famous breaches in DeFi technical security is the DAO hack[1]. In this hack, the attackers manipulated a smart contract and managed to drain funds valued at $70 million at the time. The Ethereum community intervened, resulting in a network fork to stop the attack.

Definition 4.1 (Technical Security, [21])**.** A DeFi protocol is technically secure if it is not possible for an attacker to *atomically* exploit the protocol at the expense of value held by the protocol or its users. Due to atomicity, these attacks can generate *risk-free* profit. A common property of technical exploits is that they occur within a single transaction or a bundle of transactions in a block.

The most common instances of technical breaches in security are:

- Manipulating a smart contract within a single transaction, which is risk-free for anyone. This manipulation often targets a logical error in the smart contract's design or a bug in its implementation. These kinds of attacks are atomic (i.e., can be facilitated by a single party) and risk-free because the manipulating transaction will either be valid and included in the blockchain, or invalid and discarded. In

[1] Coindesk report: https://www.coindesk.com/learn/understanding-the-dao-attack/.

the first case, the adversary profits, and in the second, they only have to pay a (relatively small) transaction fee.

- Manipulating transactions within the same block, which is risk-free for the miner generating that block.
- Bundled transactions created by an attacker that must execute atomically in the given order. The adversary can chain the transactions in such a way that they are either all executed in the given sequence, or none of them are executed. These types of attacks usually involve flash loans. Flash loans are not possible in UTxO models and are predominant in account-based models. Hence, they are outside the scope of this work.

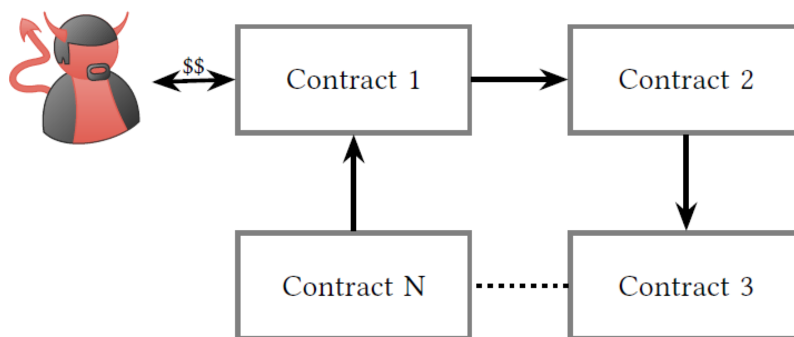

Figure 4.1: Diagram of a technical exploit, as presented in [21]. All the adversarial transactions are executed sequentially.

On the other hand, a DeFi security risk is considered economic if an attacker can perform a non-atomic exploit to profit at the expense of the protocol or its users. In an economic exploit, the attacker carries out multiple actions at different points in the transaction sequence without controlling what happens in between, meaning there's no guarantee of profit from the final action. Economic security involves an attacker manipulating a market or incentive structure over a (maybe brief) period of time,but not instantly. Unlike technical exploits, economic attacks are non-atomic, involve upfront costs, carry the risk of failure, and often require manipulation across multiple transactions or blocks.

An example of an economic attack is the Harvester incident[2], costing its users around $24M. The attacker took out a large loan in cryptocurrency and used this borrowed money to temporarily manipulate the prices of cryptocurrencies that Harvest Finance was using. This manipulation took place on other DEXs that Harvest Finance was tracking. By artificially changing the prices, they made it look like their deposits were worth much more than they actually were. The hackers then withdrew this inflated amount, repaid the loan, and kept the difference as profit. The hack was possible because of the way Harvest Finance calculated the value of its investments. The attackers took advantage of the fact that the platform relied on external sources to get

[2]Coindesk report:https://www.coindesk.com/tech/2020/10/26/harvest-finance-24m-attack-triggers-570m-bank-run-in-latest-defi-exploit/.

the prices of cryptocurrencies. By manipulating these prices for a short period, they tricked the system into giving them more money than they were supposed to get.The attack was non-atomic because it involved several coordinated steps across different DeFi protocols to manipulate the market prices. The success of the attack depended on the broader economic environment, such as the liquidity in the market and how quickly the manipulated prices could be used to drain funds from Harvest Finance.

Definition 4.2 (Economic Security, [21], adjusted)**.** A DeFi protocol is economically secure if it is economically infeasible (e.g., unprofitable) for an attacker to perform exploits that are strictly non-atomic at the expense of value held by the protocol or its users. As economic exploits are non-atomic (or else they are better described as technical), they are not risk-free. A central assumption in considering the class of economic security attacks is that of game theoretic rationality.

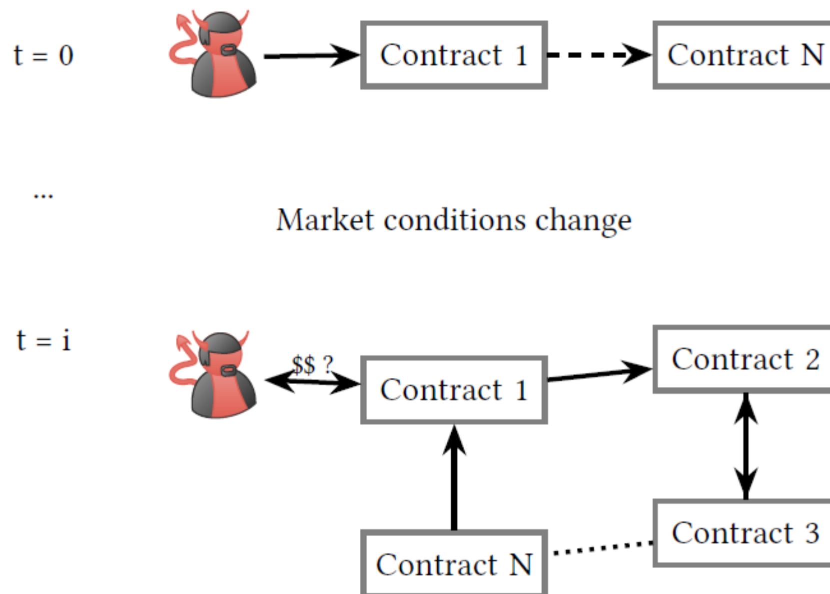

Figure 4.2: Diagram of an economical exploit, as presented in [21]. There are transactions between the attacker's transactions, reflecting the non-atomic nature of the attack.

Overall, this dichotomy clarifies the attacks in a straightforward manner. Technical attacks stem from problems that occur during the implementation of the protocol, including issues in cryptography and smart contract design. On the other hand, economic attacks exploit weaknesses in the protocol's economic model, involving issues related to its game-theoretic or financial blueprint.

4.2 Limit Order Books

Limit order books are the most common framework to facilitate trade in traditional finance. Most of the stock exchanges around the world operate under this general frame-

work. In a LOB, each party that intends to trade a specific asset pair (e.g., asset X for asset Y) submits their intent, the quantity of the asset they wish to trade, and their desired price. This type of order, which is executed only when a specified price condition is met, is known as a limit order. A collection of these limit orders forms what is known as a limit order book (LOB). The entities responsible for matching these orders are called market makers or matchmakers.

In a LOB, every trader has the option of posting buy (respectively, sell) orders. Formally:

Definition 4.3 (Limit Order Book (**LOB**), [32], Adjusted)**.** An order $x = (p_x, \omega_x, t_x)$ submitted at time t_x with price p_x and size $\omega_x > 0$ (respectively, $\omega_x < 0$) is a commitment to sell (respectively, buy) up to $|\omega_x|$ units of the traded asset at a price no less than (respectively, no greater than) p_x. A LOB $L(t)$ is the set of all active orders in a market at time t.

In each time t, there are two price thresholds that describe the state of the LOB, namely the *bid* and *ask* prices.

Definition 4.4 (Bid price, [32])**.** The bid price at time t is the highest stated price among active buy orders at time t,

$$b(t) := \max_{x \in B(t)} p_x.$$

Definition 4.5 (Ask price, [32])**.** The ask price at time t is the lowest stated price among active sell orders at time t,

$$a(t) := \min_{x \in A(t)} p_x.$$

Definition 4.6 (Bid-ask spread,[32])**.** The bid-ask spread at time t is $s(t) := a(t) - b(t)$.

At any given time t, the bid price $b(t)$ and the ask price $a(t)$ determine the thresholds for trades:

Definition 4.7 (Price changes in LOBs, [32], Adjusted)**.** Consider a buy (respectively, sell) order $x = (p_x, q_x, t_x)$ that arrives immediately after time t.

- If $p_x \leq b(t)$ (respectively, $p_x \geq a(t)$), then x is a limit order that becomes active upon arrival. It does not cause $b(t)$ or $a(t)$ to change.
- If $b(t) < p_x < a(t)$, then x is a limit order that becomes active upon arrival. It causes $b(t)$ to increase (respectively, $a(t)$ to decrease) to p_x at time t_x.
- If $p_x \geq a(t)$ (respectively, $p_x \leq b(t)$), then x is a market order that immediately matches to one or more active sell (respectively, buy) orders upon arrival. Whenever such a matching occurs, it does so at the price of the active order, which is not necessarily equal to the price of the incoming order.

Notice that, many buy orders (respectively sell orders) might be eligible for execution, but there might not be enough liquidity to fulfill each one. The responsibility for matching eligible buy orders with sell orders falls to the market maker. In the case of on-chain markets that we are interested in, this matching process can be a source of revenue for the market maker. We will discuss the consequences of this observation in the coming chapters.

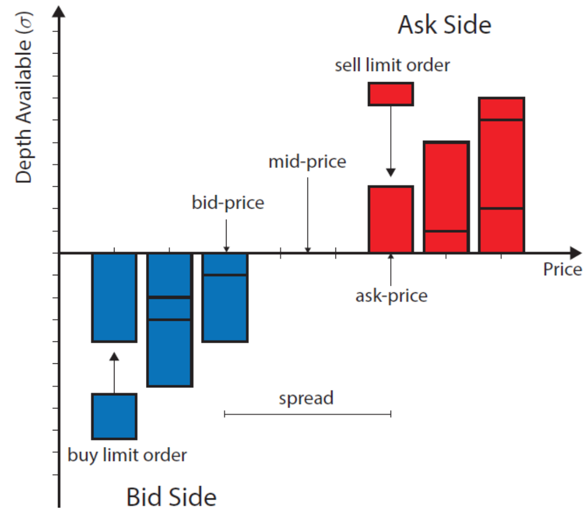

Figure 4.3: Schematic of a LOB. As given in [32]

4.3 Automatic Market Makers

Before formally defining AMMs, we have to discuss about their basic structure. Remember that an AMM is nothing more than a smart contract deployed on the blockchain to facilitate trade between users, in a permitionless and trustless manner. Also, the quoted exchange rate for an asset pair should be varifable using data that are public and stored on-chain.

Definition 4.8 (AMM actors,[33],informal)**.** Xu et al. provide an informal description of AMMs, starting with all the parties involved:

1. **Liquidity provider (LP)**: A liquidity pool can be deployed through a smart contract with some initial supply of crypto assets by the first LP. Other LPs can subsequently increase the pool's reserve by adding more of the type of assets that are contained in the pool. In turn, they receive pool shares proportionate to their liquidity contribution as a fraction of the entire pool. LPs earn transaction fees paid by exchange users. While sometimes subject to a withdrawal penalty, LPs can freely remove funds from the pool by surrendering a corresponding amount of pool shares.

2. **Exchange user (Trader)**: A trader submits an exchange order to a liquidity pool by specifying the input and output asset and either an input asset or output asset quantity; the smart contract automatically calculates the exchange rate based on the conservation function as well as the transaction fee and executes the exchange order accordingly.

3. **Protocol foundation**: A protocol foundation consists of protocol founders, designers, and developers responsible for architecting and improving the protocol.

The development activities are often funded directly or indirectly through accrued earnings such that the foundation members are financially incentivized to build a user-friendly protocol that can attract high trading volume.

Definition 4.9 (AMM Assets, [33], informal, adjusted)**.** Several distinct types of assets are used in AMM protocols for operation:

1. **Risky assets**: Characterized by illiquidity, risky assets are the primary type of assets for which AMM-based DEXs are designed. Like centralized exchanges, an AMM-based DEX can facilitate an initial exchange offering to launch a new token through liquidity pool creation, which is particularly suitable for illiquid assets. Note that this category includes all the secondary assets described in the introduction.
2. **Base assets**: This is the native token of the blockchain on which the AMM is deployed. Some protocols require a trading pair to always consist of a risky asset and a designated base asset. Most of the time, its price is more stable than that of risky assets, due to the native token's overall utility in the blockchain.
3. **Pool shares**: Also known as “liquidity shares” and “LP shares,” pool shares represent ownership in the portfolio of assets within a pool and are distributed to LPs. Shares accrue trading fees proportionally and can be redeemed at any time to withdraw funds from the pool.
4. **Protocol tokens**: Protocol tokens represent voting rights on protocol governance matters and are thus also referred to as “governance tokens”. Protocol tokens are typically valuable assets that are tradable outside of the AMM and can incentivize participation when, for example, rewarded to LPs proportionate to their liquidity supply. AMMs compete with each other to attract funds and trading volume. To bootstrap an AMM in the early phase with incentivized early pool establishment and trading, a feature called liquidity mining can be implemented, where the native protocol’s tokens are minted and issued to LPs and/or exchange users.

In this work, the only relevant type of assets are the risky ones and the base asset, which we will otherwise call *native token* or *numéraire* [3]. We will use those terms interchangeably. Additionally, in this work, we are primarily focused on risky assets and the numéraire, and will steer away from discussing LP shares and governance tokens. These have been included only for completeness and to provide the reader with a more comprehensive view of the AMM mechanism.

Definition 4.10 (Fundamental AMM dynamics, [33], informal,adjusted)**.** .

1. **Invariant properties**: The functionality of an AMM depends upon a conservation function which encodes a desired invariant property of the system. As an intuitive example, UniSwap’s constant product function determines trading dynamics between assets in the pool as it always conserves the product of value-weighted quantities of both assets in the protocol—each trade has to be made

[3] In Finance: The numéraire is often used as a benchmark to express prices or values of various assets. It might be a specific currency like USD or a blockchain's native token.

in a way such that the value removed in one asset equals the value added in the other asset. This weight-preserving characteristic is one desired invariant property supported by the design of UniSwap.

2. **Mechanisms**: An AMM typically involves two types of interaction mechanism: asset swapping of assets and liquidity provision/withdrawal. Interaction mechanisms have to be specified in a way such that desired invariant properties are upheld; therefore the class of admissible mechanisms is restricted to the ones which respect the defined conservation function, if one is specified, or conserve the defined properties otherwise.

4.3.1 Constant Product Automatic Market Makers

Now that we are familiarized with the basic concepts behind AMMs, we will analyze the most basic and popular instance of AMMs, namely *Constant Product Market Makers* (**CPMM**). We will begin by stating the basic form of a CPMM with no trading fees, as seen in [34] by Zhang et al. Let us suppose that this AMM is facilitating the trade of the X-Y token pair.

Let x and y be the number of tokens X and Y, respectively, that the exchange currently holds. We will call this number the *reserve* or the *liquidity* of a single token in the pair. In the constant product AMM implementation, as the name suggests, we set the exchange price between the two tokens so that the product $x \times y$ remains constant.

So after a sell order of token X in this pair, i.e. Δx tokens X for Δy tokens of Y. The constant product rule must hold, hence:

$$x \cdot y = (x + \Delta x) \cdot (y - \Delta y)$$

Thus, the price $\frac{\Delta x}{\Delta y}$ is a function of $\frac{x}{y}$.

The new reserves are updated:

$$x' = x + \Delta x = (1 + \alpha)x = \frac{1}{1 - \beta}x$$

$$y' = y - \Delta y = \frac{1}{1 + \alpha}y = (1 - \beta)y$$

where

$$\alpha = \frac{\Delta x}{x}, \beta = \frac{\Delta y}{y}$$

We can calculate the amounts Δx and Δy by solving for them. Hence:

$$\Delta x = \beta \frac{1}{1 - \beta}x, \Delta y = \alpha \frac{1}{1 + \alpha}y$$

Definition 4.11 (AMM state, [35])**.** The state (or depth) of an AMM market X/Y at time t is defined as $s_t = (x, y)$, where x is the amount of asset X, and y is the amount of asset Y in the liquidity pool. The state at a given blockchain block N is denoted as $s_N = (x_N, y_N)$.

Each trade in a CP AMM impacts the AMM's state.

A trade on a CPMM

The liquidity pool initially holds 100 units of Token X and 200 units of Token Y, where x and y represent the reserves of Token X and Token Y, respectively. In this scenario, k is given by:

$$k = 100 \times 200 = 20{,}000$$

The price of Token X in terms of Token Y before any trade occurs is:

$$P_{\text{AB}} = \frac{y}{x} = \frac{200}{100} = 2 \text{ Token Y per Token X}$$

Let us suppose that Alice wants to trade 10 tokens X. After this trade, the reserves of Token X increase to:

$$x' = 100 + 10 = 110$$

To maintain the constant product k, the new reserve of Token Y, denoted by y', must satisfy:

$$110 \times y' = 20{,}000 \Longrightarrow y' = \frac{20{,}000}{110} \approx 181.82$$

Thus, Alice receives:

$$\Delta y = 200 - 181.82 = 18.18 \text{ Token Y}$$

The effective price realized by the trader, considering the trade's impact, is:

$$\text{Effective Price} = \frac{\Delta y}{\Delta x} = \frac{18.18}{10} \approx 1.82 \text{ Token Y per Token X}$$

The example above illustrates the impact that a trade has on an AMM's liquidity, and the trader's return. This phenomenon is known as slippage.

Definition 4.12 (Slippage, [21])**.** Slippage is defined as the difference between the spot price and the realized price of a trade. Instead of matching buy and sell orders, AMMs determine exchange rates on a continuous curve, and every trade will encounter slippage conditioned upon the trade size relative to the pool size and the exact design of the conservation function. The spot price approaches the realized price for infinitesimally small trades, but they deviate more for bigger trade sizes. This effect is amplified for smaller liquidity pools as every trade will significantly impact the relative quantities of assets in the pool, leading to higher slippage.

Definition 4.13 (Liquidity sensitivity, [33] [36], adjusted)**.** An AMM is liquidity-sensitive when a fixed swap size (same input quantity) makes a larger price impact, i.e. higher slippage, in a thin liquidity pool than a deep liquidity pool.

Definition 4.14 (Demand sensitivity, [33])**.** An AMM is demand-sensitive when the average swap price (i.e. the effective exchange rate) increases as the swap size (input quantity) increases. Intuitively, this suggests that as with the increment of the demand in output token, its price denominated input token will be driven up.

A constant product AMM is both liquidity-sensitive and demand-sensitive. Hence, The relative size of the trade in relation to the AMM's liquidity is crucial for determining the trade's effective price. In the next chapter, we will discuss how transactions with significant slippage can be targeted by profit-maximizing bots to generate profit, often resulting in a worse effective price for the trade.

A geometric interpretation

Since trading on a Constant Product (CP) AMM maintains the product of the two tokens as a constant, its geometric representation is a hyperbola, where each point represents a possible liquidity state of the X-Y token pair. If the AMM is at point A, then the coordinates of the vector $\vec{AB}$ represents the amount a trader must pay in one token to receive a corresponding amount of the other token. This visualization makes it easy for a trader to compute the outcome of a trade, making a CP AMM an intuitive and straightforward trading mechanism.

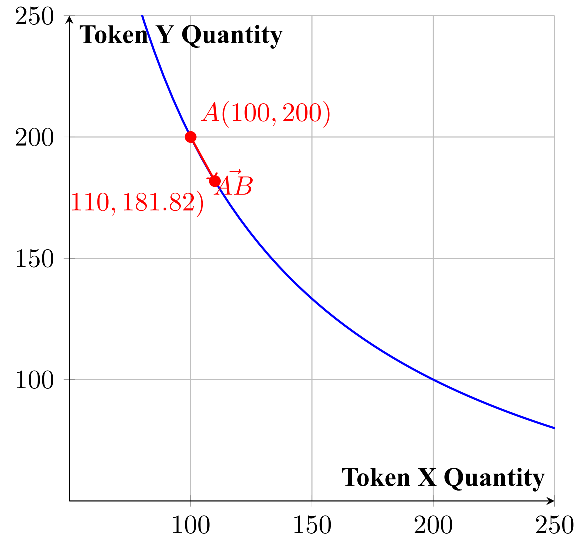

Figure 4.4: The graph of the CP AMM given in our example

A δ_x, δ_y trade on a CPMM[34] market with state (liquidity) $s = (x, y)$ depends only on the size of the trade. Partitioning the trade on smaller ones does not change the outcome of the trade (for better or for worse). This property in known as Path Independence [33].

Proof for 2 trades:

Let $X \cdot Y = k$ a CPMM with state $s = (x, y)$ at time t. A trader swaps δ_x of token X on it for δ_y tokens Y.

1-shot execution of this trade:

$$(\delta_x + x)(y - \delta_y) = k \Rightarrow$$

$$(y - \delta_y) = \frac{k}{(\delta_x + x)} \Rightarrow$$

$$\delta_y = y - \frac{k}{(\delta_x + x)}$$

2-shot execution of this trade:

With inputs $\delta_{x_1}, \delta_{x_2}$ and outputs $\delta_{y_1}, \delta_{y_2}$ respectively, such that $\delta_{x_1} + \delta_{x_2} = \delta_x$. We want to prove that $\delta_{y_1} + \delta_{y_2} = \delta_y$

First trade within input δ_x:

$$(\delta_{x_1} + x)(y - \delta_{y_1}) = k \Rightarrow$$

$$(y - \delta_{y_1}) = \frac{k}{(\delta_{x_1} + x)} \Rightarrow$$

$$\delta_{y_1} = y - \frac{k}{(\delta_{x_1} + x)}$$

After the execution, the state of the CPMM is updated $s' = (x', y') = (x + \delta_{x_1}, y - \delta_{y_1})$

Second trade execution:

$$(\delta_{x_2} + x')(y' - \delta_{y_2}) = k \Rightarrow$$

$$(y' - \delta_{y_2}) = \frac{k}{(\delta_{x_2} + x')} \Rightarrow$$

$$\delta_{y_2} = y' - \frac{k}{(\delta_{x_2} + x')}$$

We substitute x', y' and we get:

$$\delta_{y_2} = y - \delta y_1 - \frac{k}{(\delta_{x_1} + \delta x_2) + x} \Rightarrow$$

$$\delta_{y_2} + \delta y_1 = y - \frac{k}{(\delta_{x_1} + \delta x_2) + x} \Rightarrow$$

$$\delta_y = y - \frac{k}{(\delta_x + x)}$$

So we notice that the 2-shot serial execution of a trade gets the exact same returns as the 1-shot execution. This argument can be generalized for n-shot serial executions with induction.

4.3.2 UniSwap

In the previews section, we discussed the basic scheme for a CP AMM, but we have abstracted away one important detail: trading fees. As mentioned earlier, an AMM requires liquidity to function, which is provided by liquidity providers (LPs). Therefore, the AMM must offer financial incentives to encourage LPs to deposit liquidity. These incentives come from the AMM's trading fees ρ, which are applied to each transaction as a small percentage of the transaction's value. This model is the first version of Ethereum's flagship DEX, UniSwap [34].

We have three parameters:

$$\alpha = \frac{\Delta x}{x}, \beta = \frac{\Delta y}{y}, \gamma = 1 - \rho$$

After a X sell order (equivalent to the previews example), the new reserves of the AMM will be:

$$x' = x + \Delta x = (1 + \alpha)x = \frac{1 + \beta(\frac{1}{\gamma} - 1)}{1 - \beta}x$$

$$y' = y - \Delta y = \frac{1}{1 + \alpha\gamma}y = (1 - \beta)y$$

The cost (denominated in token X) will be $\Delta x = \frac{\beta}{1-\beta} \cdot \frac{1}{\gamma} \cdot x$.

The return (denominated in token Y) will be $\Delta y = \frac{\alpha\gamma}{1+\alpha\gamma} \cdot y$.

Naturally, the numerical example and the geometric analysis we provided for the CPMM without fees extend to UniSwap, albeit with slight changes in values due to the trading fees. UniSwap's trading fees are approximately 0.3%. .

4.4 LOB and CPMM comparison

The most important aspect of DEX mechanisms is how they manage liquidity, i.e., the existing and future supply of the risky asset (numéraire) under their jurisdiction. In the LOB paradigm, things were simple. Trades in a pair occurred only when both parties were satisfied, i.e., when both the buyers and the sellers had quoted prices within the range of their corresponding accepted prices. This property is advantageous, but the mechanism does not perform well under extreme market conditions, which are not uncommon in the DeFi world.

Suppose we have an LOB managing the X-Y trading pair, with X being the numéraire and Y being the risky asset. If, for whatever reason, there are extremely few sell orders for token Y in the market and a lot of buying demand, i.e., high demand for token Y, basic economic principles suggest that the price of token Y, denominated in token X (the numéraire), will increase.

Using an LOB to implement this market presents a few problems:

1. Token Y sellers must monitor the market and adjust their prices accordingly, or they risk selling their valuable token at a lower price. Tracking blockchain market conditions in real-time is generally difficult and requires specialized equipment and technical knowledge.

2. Even if sellers take the effort to update their sell prices, this process can be costly due to the substantial fees involved. We will not assess whether this is still advantageous for them, but this complexity adds an additional layer of difficulty for users.

3. Classical LOB orders must be completed as stated or not at all. Therefore, a trader using LOBs is guaranteed to receive the price they wish, but they might have to wait until the bid and ask prices adjust accordingly for their trade to be eligible for execution.

On the other hand, CPMMs are designed to ensure that liquidity never runs out and that prices adjust according to natural market conditions (i.e., supply and demand). Additionally, trades are executed nearly instantly. However, due to the constantly shifting prices in the CPMM model, traders might experience suboptimal outcomes.

These observations are formally captured and proven by Roughgarden et al. in [37]. They proved that the expressiveness of an on-chain trading mechanism is proportional to its complexity, and thus the amount of on-chain data it requires to function. Therefore, although CPMMs may be less descriptive than LOBs, their state can be more easily saved on-chain, which reduces the trading fees required to operate.

Conversely, LOBs require more on-chain data space to save their state. However, they are far more descriptive than CPMMs, ensuring more accurate and satisfactory outcomes for traders.

CHAPTER 5 TRANSACTION ORDERING

The notion of transaction ordering within the same block seems straightforward, but it may hold caveats when considering the role that the blockchain plays as a foundation for different DeFi applications. As we discussed in the previous chapter, each transaction on a CPMM alters its state and thus impacts the price between the pair being traded on that DEX. Therefore, the order of transactions within the same block can result in vastly different outcomes for the trades these transactions encode.

Perhaps the most illuminating example of this occurs during times of extreme market volatility. Let's suppose that, for whatever reason (e.g., a hack taking place), there is significant selling pressure on asset X in the X-Y pair. This means that all the X sell orders will negatively impact X's price. Therefore, the order of these transactions will directly influence each transaction's outcome. Each trader, being rational, will have an incentive to have their transaction prioritized within that block. Since the difference in price can be substantial, each trader has an incentive to tip the BP to prioritize their transaction. The BP plays the role of the kingmaker in this scenario, so each trader will compete by offering increasing amounts for priority. This creates a kind of auction for block space, which is a well-known phenomenon in Ethereum known as *priority gas auctions*.

On the other hand, the Cardano blockchain does not include an option for users to tip a BP for prioritizing their transactions, at least not through an on-chain mechanism. Thus, the same scenario that can earn an Ethereum BP a substantial reward as a tip is irrelevant for a Cardano BP. It becomes clear that when discussing the financial nature of transaction ordering, the underlying blockchain infrastructure plays a pivotal role in this discussion.

Definition 5.1 (Domain, [38])**.** A domain D is a self-contained system with a globally shared state st. This state is altered by various agents through actions (sending transactions, constructing blocks, slashing, etc.), that execute within a shared execution environment's semantics. Each domain has a predefined consensus protocol that includes a set of valid algorithms to order transactions, denoted by **prt(D)**.

Each individual blockchain is a domain. However, there are other non-blockchain domains, like decentralized exchanges that execute their transactions off-chain. Most of the body of work around transaction ordering, including transaction reordering attacks for profit maximization, focuses on Ethereum. That will be our focus for the rest of the chapter, beginning with a few key definitions, as given by Daza et al. in [38].

5.1 Rational Block Producers

Definition 5.2 (Types of Block Producers, [38], adjusted)**.** We distinguish four types of BPs:

1. **Dummy**: A BP is *dummy* if he follows the validator consensus protocol **prt(D)** as indented.
2. **Dummy Byzantine**: A BP is *dummy Byzantine* if they misbehave, but other nodes can detect his misconduct. This notion is akin to an adversarial BP in classic distributed protocols.
3. **Rational**: A BP is *rational* if it follows a set of valid actions on the domain D to maximize its utility function, including deviating from **prt(D)**. Therefore, if a player misbehaves to maximize their payoff but cannot be identified and punished, we say that is a rational player.
4. **Partially Rational**: A sequencer is *partially rational* if they commit to using a specific valid ordering mechanism to maximize its payoff.

For now, we will assume that each BP is partially rational, meaning that they construct a block (i.e., a valid sequence of transactions) in a way that maximizes their tipping rewards. However, they do not include their own transactions to further increase their utility, which distinguishes them from a fully rational BP.

Definition 5.3 (Knapsack Extractable Value (**KEV**) Problem, [38], adjusted)**.** A BP receives a set of transactions $tx_1, \ldots, tx_n$ with gas prices $m_1, \ldots, m_n$ and $g_1, \ldots, g_n$ units of gas. If the sequencer includes tx_i, it obtains $m_i g_i$ in fees. Since the gas used per block is restricted in every domain by some constant L, the sequencer must choose a subset of transactions T such that

$$\sum_{tx_i \in T} g_i \leq L.$$

Then, a node that tries to maximize its revenues per block needs to solve the following Knapsack optimization problem:

$$\begin{aligned} \max \sum_{i=1}^{n} x_i m_i g_i \\ \text{s.t.} \sum_{i=1}^{n} x_i g_i \leq L, \\ x_i \in \{0, 1\}. \end{aligned}$$

Daza et al. note that since Knapsack optimization problems are NP-complete, each sequencer usually chooses different algorithms to approximate the optimal solution.

5.2 Transaction Reordering Attacks

We have seen that BPs are responsible for determining the order in which transactions appear within a block. If a BP is dummy, transactions should be ordered in the sequence in which the BP receives them, i.e., as they appear in the BP's mempool. However, if a BP acts rationally, this may not always be the case. It's important to note that as long as the BP operates within the rules of the consensus mechanism, they are not considered adversarial. Instead, they are exploiting the authority granted to them to extract more value for themselves. These practices are knows as *transaction reordering manipulations*.

Definition 5.4 (Transaction reordering manipulations, [39])**.** There are three kinds of transaction reordering required by the most common BPS opportunities:

1. **Fatal front-running**: Fatal front-running describes a transaction reordering manipulation by which the attacker's transaction T_A front-runs (executes before) the victims transactions T_V . In the process, the attacker's transaction causes the victim's transaction to fail.

2. **Front-running**: Front-running is a transaction reordering manipulation that has attacker's transaction T_A front-run the victims transactions T_V. As opposed to fatal front-running, the attacker ensures that the victim transaction will still execute. Generally, the conditions for the victim transaction, will however, be worse.

3. **Back-running**: Back-running occurs when the attacker's transaction T_A back-runs (executing after) the victims transactions T_V.

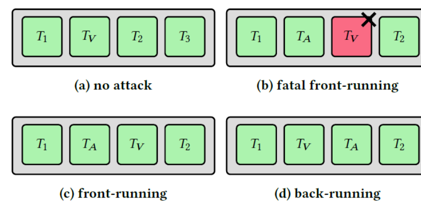

Figure 5.1: Figure (a) shows a block with no manipulation, while Figures (b), (c), and (d) illustrate different transaction reordering strategies. Successful transactions are green, and failed transactions are red with a cross. As presented in [39].

5.2.1 Sandwich Attacks

As the name suggests, sandwich attacks are an advanced transaction reordering strategy, combining front-running and back-running transactions to envelop (i.e., 'sandwich') a user's transaction. Sandwich attacks are a common strategy to extract Block Producer Surplus from trades on a Constant Product Automated Market Maker (Section 4.3.2).

As discussed earlier, a relatively large to the liquidity depth enough X-Y trade will cause slippage in an AMM pair. This slippage can be calculated by any party that knows the initial state $s_0 = (x_0, y_0)$ of the AMM before a transaction and the details

of the transaction T_V (e.g., selling an amount of asset X for asset Y). Both pieces of information are available to a BP whenever a transaction is submitted to their mempool for block inclusion. A BP can then execute the following strategy, creating and ordering transaction on his block:

- Create a buy order, T_F, to inflate the price of asset Y.
- Include the trader's buy order, T_V, further inflating the price of token Y.
- Finally, include their sell order, T_B, selling all the units of asset Y acquired from T_F.

Since the trader's buy order T_V is executed between the BP's buy order T_F and sell order T_B, the per-unit price of asset Y has increased. Additionally, because T_V was targeted for creating significant slippage, it is relatively large compared to the AMM's liquidity depth, making the increase in Y's per-unit price substantial. Hence, the BP profits from this series of transactions. The proof of this claim is merely an algebraic application of the CP AMM's equations, and will be omitted.

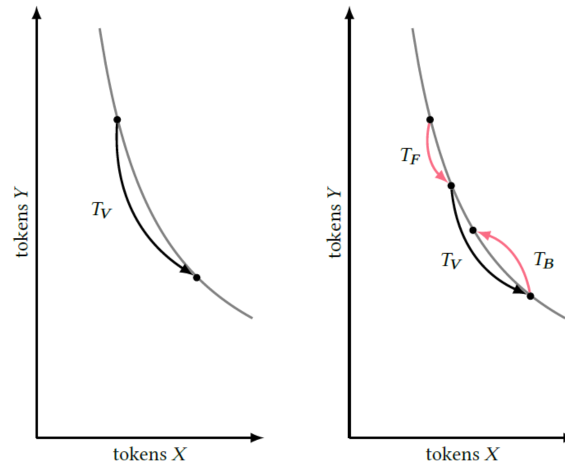

Figure 5.2: Sandwich attack visualization in a CP AMM. Notice that T_V receives a worse price in the presence of the sandwich attack. As presented in [39].

5.2.2 Cyclic Arbitrage Opportunities

CPMMs can create cyclic arbitrage opportunities due to temporary price inaccuracies across liquidity pools. For instance, consider a CPMM with three liquidity pools between assets X, Y, and Z. If prices are unsynchronized, a trader can potentially profit by executing a cyclic trade: exchanging X for Y at price $P_{X\to Y}$, then Y for Z at price $P_{Y\to Z}$, and finally Z for X at price $P_{Z\to X}$. The trade is profitable if:

$$P_{X\to Y} \cdot P_{Y\to Z} \cdot P_{Z\to X} \geq 1.$$

This opportunities can be the target of a transaction reordering attack by the following two ways:

1. **Fatal front-running**: When a user finds such an opportunity and submits a corresponding transaction to the mempool, anyone listening to the mempool can subsequently see this arbitrage opportunity. Through fatal front-running, an attacker can steal such an arbitrage opportunity.

2. **Back running**: The attacker can predict the imbalance just as it is created by a transaction. Then, they just need to back-run this transaction to capture all the arbitrage profit.

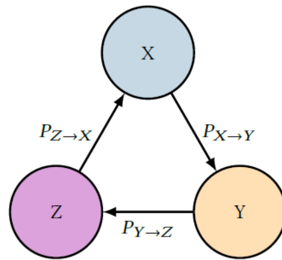

Figure 5.3: Cyclic trade execution between tokens X,Y and Z. As presented in [39].

5.2.3 Front-running in the eUTxO Model

As Heimbach and Wattenhofer stated in [39], transactions in the UTxO and eUTxO models cannot be front-run. Instead, any front-running attempt by an adversary would cause the transaction to fail.

The outline of the proof behind this claim is as follows: Suppose the victim's transaction T_V references $UTxO_1$ as its input. To front-run the victim transaction, the adversary must create a front-running transaction T_F that also references $UTxO_1$ as its input, and have T_F execute before T_V. If successful, $UTxO_1$ would be spent, causing the subsequent transaction T_V to become invalid and fail.

Notice that the front-running attempt fails because the adversary cannot force T_V to reference T_F's UTxO as its input. This property is a byproduct of the eUTxO model's requirement for specific input references. In contrast, the account-based model does not specify inputs for each transaction, treating them instead as changes in account balances. This distinction clearly highlights the difference between the eUTxO and account-based models.

This is an important property of the eUTxO model, which also applies to any transaction reordering attacks involving front-running transactions. Therefore, in the eUTxO model, transactions cannot be sandwiched either.

However, this desirable property does not hold in Cardano DeFi. In the next chapter, we will see that due to concurrency issues (subsection 3.3.2) in the eUTxO model, DeFi applications cannot achieve the necessary throughput to meet users' demand for trades.

As a result, DEXs resort to off-chain processing and execution of transactions, where the aforementioned property of the eUTxO model no longer applies.

5.3 Block Producer Surplus

Miner Extractable Value (**MEV**), or *Block Producer Surplus* (**BPS**), as it later came to be known, refers to the profits that block producers earn by leveraging their ability to order transactions. This can include arbitrage opportunities as well as additional fees that users may pay to BPs to prioritize their transactions within a block. Assuming rational participation in the protocol (i.e., block producers being profit maximizers), nodes will seek to exploit any MEV opportunities to maximize their payoff.

MEV opportunities arise from the *information asymmetry* between users and block producers. Since BPs *choose* and *order* the transactions contained in each block, they can profit by including or excluding transactions as they see fit.

Daian et al. in [40] were the first to define BPS and analyze the opportunities that block producers exploit to harvest it. In their work, they explored design flaws in decentralized exchanges that led to the emergence of arbitrage bots and measured their behavior. They identified that most BPS is extracted from the *direct* or *indirect* exploitation of Pure Revenue Opportunities.

Definition 5.5 (Pure Revenue Opportunities (**PROs**), [40], Adjusted)**.** A specific subcategory of DEX arbitrage representative of broader activity, these are blockchain transactions that issue multiple trades atomically through a smart contract and profit unconditionally in every traded asset.

Sandwich attacks and Cyclic Arbitrage Opportunities are instances of PROs. As the name suggests, when located in a BP's mempool, PROs offer atomically executable and risk-free profit to either the BP or any other entity that can take advantage of this opportunity by launching a successful transaction reordering attack.

In practice, in the Ethereum ecosystem, the aforementioned pure revenue opportunities are identified by bots, known as *searchers*, rather than BPs. These bots engage in bidding wars, each offering increasingly higher transaction fees to the BP so that they can be the searcher to capture the PRO. These phenomena are called Priority Gas Auctions.

Definition 5.6 (Priority Gas Auctions (**PGAs**), [40], Adjusted)**.** Because pure revenue opportunities offer unconditional revenue, arbitrage bots compete against each other by bidding up transaction fees (gas) in what we call PGAs.

Naturally, BPs profit from these auctions. These profits are called *Ordering Optimization* (**OO**) fees. This phenomenon is well known in Ethereum, and it has been officially adapted in the protocol. It is called *searcher-builder separation*, as discussed in [41].

Summing up, in the presence of PROs, a BP can profit in two ways. Either by directly launching a transaction reordering attack themselves to harvest this arbitrage

opportunity, or by harvesting the Ordering Optimization fees stemming from the Priority Gas Auctions. Either way, this is reflected in the following definitions of BPS, given by Roughgarden et al.

Definition 5.7 (Transaction Fee Mechanism (**TFM**), [42], Adjusted)**.** A transaction fee mechanism is a triple (x, p, q) where:

- x is a allocation rule,
- p is a payment rule. It is a function p that specifies a nonnegative payment $p_t(B, b)$ for each transaction $t \in B$ in a block B, given the bids b of all known transactions.
- q is a burning rule. It is a function q that specifies a nonnegative burn $q(B, b)$ for a block B, given the bids b of all known transactions.

Each BP values a transaction according to his model. Roughgarden et al. identify two discrete cases:

1. Additive valuation: An additive valuation corresponds to a BP that extracts value from each transaction independently. $v_{BP}(B) := \sum_{t \in B} \mu_t$
2. Single-minded valuation: This corresponds to an PRO exploitation. It is either equal to 0 (if the BP does not capture the liquidation opportunity in B) or to some fixed constant μ (otherwise, where μ is the value of the opportunity).
$$v_{BP}(B) = \begin{cases} \mu & \text{if } B \in S \\ 0 & \text{otherwise} \end{cases}$$

Definition 5.8 (Block Producer Surplus (BPS), [42], Adjusted)**.** For a TFM (x, p, q), BP valuation v_{BP}, BP blockset B (the non-empty set of all possible blocks) , and transaction bids b, the block producer surplus of a BP that chooses the block $B \in B$ is

$$u_{BP}(B) := v_{BP}(B) + \sum_{t \in B} p_t(B, b) - q(B, b).$$

We remark that, as noted in [38], the blockset of each BP is constrained by that player's information (mempool), gas efficiency, budget, and ability to propose blocks. Hence, at any given time, each BP has a unique BPS associated with it.

One final notion of BP utility that will be relevant later in this work is the Off-Chain Agreement, as introduced by Roughgarden in [43]. Simply put, each creator of a transaction t agrees to submit t with an on-chain bid of b_t while transferring τ_t to the miner m off-chain; the miner, in turn, agrees to mine a block comprising the agreed-upon transactions of T.

5.3.1 Off-Chain Agreement

Definition 5.9 (Off-Chain Agreement (**OCA**), [43], Adjusted)**.** For a miner m and a set T of transactions, an off-chain agreement between T''s creators and m specifies:

- a bid vector **b**, with b_t indicating the bid to be submitted with the transaction $t \in T$;

- an allocation vector **x**, indicating the transactions that the miner m will include in its block;
- a transfer τ_t from the creator of each transaction $t \in T$ to the miner m.

Although instances of OCA are not generally known in Cardano at the moment, it is prudent to account for them in a DeFi protocol, as it is in the best interest of colluding users and BPs to maintain this secrecy.

CHAPTER 6

EUTXO DEFI

6.1 Transaction Batchers

As discussed in Subsection 3.3.2, concurrency is a significant issue affecting the scalability of a DEX protocol within the eUTxO accounting model. Since each UTxO can be referenced only once per block, each smart contract UTxO can be accessed by only one user per block. Given Cardano's block production speed of one block every 20–40 seconds, this creates a bottleneck for popular DeFi applications, such as CPMMs and LOBs.

To address this challenge, Cardano's ecosystem introduced a novel primitive called the *batcher*. Batchers mitigate the concurrency issue by aggregating all transactions referencing the same eUTxO into one, and then including this aggregate transaction within a block. As a result, the batcher's UTxO and the DEX's UTxO are referenced together in a single transaction, with the outputs distributed to the users who initially interacted with the batcher. Below, we provide a more detailed analysis of the batcher primitive, drawing from the information in [44]. To the best of our knowledge, there is no formal work analyzing the concept of UTxO transaction aggregation via batchers.

Batchers are entities that run special nodes on the Cardano network. While many batchers may also be the same entities as Cardano BPs, this is not necessary. What all batchers have in common is the ability to create and submit transactions that invoke the validator scripts of the UTxOs belonging to a DEX's smart contracts. Batchers aggregate multiple user transactions into a single transaction and submit it to a BP to be included in a future block.

The Batching Process

The batching process works in two steps. In the first step, users submit their transactions by locking their funds in a smart contract controlled by the batcher. They also specify the conditions under which their funds can be spent. In an X-Y trading pair, this would be the amount of token Y they expect to receive in exchange for the amount of token X they locked in the batcher's smart contract. This smart contract UTxO can

be spent by anyone who creates a valid transaction that satisfies those conditions, but it is usually spent by the batcher.

In the second step, the batcher creates a single transaction. Its inputs consist of all the users' UTxOs that are locked in the batching contract and are eligible for execution. The eligibility is determined by the DEX's economic model (either LOB or CPMM) and is specific to each DeFi application. Additionally, the transaction takes as input the DEX's UTxO. All these UTxOs are aggregated together and are submitted as a single atomic transaction, meaning they are either all executed, on non is executed at all.

The transaction outputs are UTxOs belonging to each user who submitted a transaction in the first step, the DEX's UTxO to be referenced in future transactions, and a UTxO belonging to the batcher that contains their batching fees.

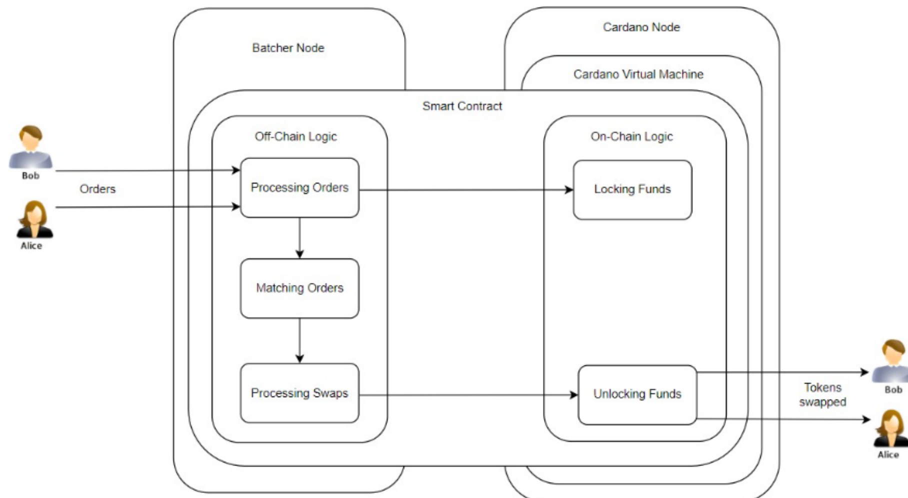

Figure 6.1: The Batching Process, as given in [44].

A batcher cannot use the funds locked into their smart contract for transactions that do not satisfy the users' spending conditions. Hence, they cannot divert funds in any way and must use them to fulfill each user's demands. However, they have the power to order transactions in any way they see fit. This enables them to orchestrate transaction reordering attacks, bypassing the front-running protection offered by the eUTxO model.

Additionally, batchers represent a single point of failure for any DEX that utilizes them, weakening the technical security of the DeFi protocol and opening up additional attack vectors. We will discuss an example of batcher manipulation later in the chapter.

Lastly, batchers are entities that are either trusted by a DEX or directly controlled by the DEX team. This raises centralization concerns and encourages monopolistic behavior within Cardano's DeFi ecosystem.

6.2 MuesliSwap

MuesliSwap [45] was the first DEX to be deployed on the Cardano main net, launching on November 2021. It operates under the Limit Order Book model described in 4.2. Given a specific X-Y trading pair, there are two types on entities interacting on the model, *traders* and *matchmakers*.

- **Trader:** A trader interacts with the LOB by placing *limit orders* via a MuesliSwap's batching smart contract. The limit order locks the tokens the trader is willing to sell in a UTxO, along with a fee in ADA. The limit order is denoted as $O = (X, Y, n, p, f)$, where:
  - X: The token the trader is willing to sell,
  - Y: The token the trader wants to buy,
  - n: The quantity of token Y the trader wants to buy,
  - p: The maximum price of X per unit of Y,
  - f: The batching fee in ADA locked with the order.

  The order's UTxOs remains locked on the batching smart contract until it is either executed or removed by the trader.

- **Matchmaker:** Matchmakers monitor the batching smart contract for *executable* orders. Suppose they find n orders $O_1, \ldots, O_n$ that:
  - Are ready for execution.
  - Have sufficient liquidity in both directions to cover at least $n - 1$ orders fully.
  - Satisfy the condition $\sum_{i=1}^{n} f_i > F$, where F is the network fee for the aggregate order.

  If all the above conditions are met, the matchmaker batches together all the relevant UTxOs and executes an atomic swap, satisfying all parties and retaining $\sum_{i=1}^{n} f_i - F$ ADA as fees.

Note that there may not be enough liquidity in one direction to fully fulfill order O_n. In that case, a new UTxO will be created for the unfulfilled remainder and locked in the batching smart contract for future fulfillment.

In [45], the MuesliSwap team notes that a decentralized order book relying on a single matchmaker to coordinate exchanges may exhibit monopolistic behaviors, such as excluding certain traders. They emphasize the need for multiple matchmakers to be active in the protocol and aim to create a market for matchmakers using incentives to ensure that matching and ordering are executed fairly.

However, in [46], they state that all matchmakers currently active on MuesliSwap are licensed by the MuesliSwap team. This restriction is enforced by allowing only licensed entities to interact with the validator scripts that secure the DEX's UTxOs, effectively preventing any other entity from serving as a matchmaker. We note that this monopolistic behavior by the MuesliSwap team has created issues for the DEX, which we will explore shortly. Hence, although the project's whitepaper does not directly state this, MuesliSwap falls under the *Trusted Third Party Transaction Ordering*, as described in [39] by Heimbach and Wattenhofer.

6.2.1 BPS incidents in MuesliSwap

Although not clearly stated in the whitepaper, MuesliSwap allows users to specify a maximum slippage ratio in their transactions, i.e., given an $X - Y$ sell order, a percentage over the maximum price of X per unit of Y. Thus, given an order $O = (X, Y, n, p, f)$ with slippage s, the slippage can be incorporated into the order and encoded as $O' = (X, Y, n, p', f)$, such that $p' \in (p, \frac{100p+s}{100})$, matching MuesliSwap's order format.

However, if the order O is in the opposite direction (i.e., an $X - Y$ buy order), the slippage s will affect the price in the opposite way. Hence, the order becomes $O' = (X, Y, n, p', f)$, where $p' \in (\frac{100p-s}{100}, p)$.

Note that this notion of slippage is not the same as given in Definition 4.12. Slippage is a notion relevant to CPMMs and is not applicable to a LOB such as MuesliSwap.

This results in two bid prices: the original bid price as intended by the trader, denoted b, and the bid price after slippage, denoted b'. Note that if slippage is non-zero, $b < b'$. Similarly, trades in the opposite direction will affect the ask price a, resulting in two distinct ask prices, a and a', where $a' < a$ when $s \neq 0$. We will update the standard LOB notation to much this implementation, denoting the spread as usual, and defining the spread after slippage to be the difference of the bid and ask prices after slippage. See figure 6.2 for a visualization.

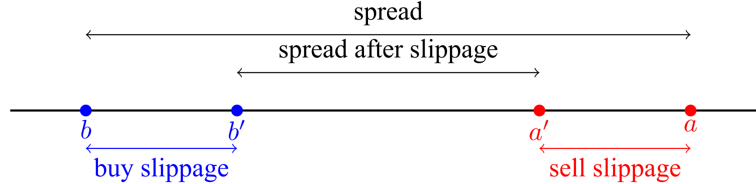

Figure 6.2: A visualization of the spread and spread after slippage values at a given LOB.

At the beginning of August 2023, the Cardano community noticed[1] that swaps on the DEX were executed using orders at their maximum slippage price p', in *both directions*. Although we do not have on-chain data from that time to verify the validity of this claim, we will assume it is true, as it has not been disputed by the MuesliSwap team in their report on the incident.[2].

We assert that this is an act of Miner Extractable Value by the matchmaker, by proving that a rational matchmaker that was the option of executing a swap with a slippage margin, will always choose to execute the swap at max slippage. To do so, we only have to prove this claim for orders that get matched to an active order as soon as they join the LOB. These events are the only ones that remove orders from the order book, and are thus the only ones that generate any profit for the matchmaker.

[1] The issue was raised via X `https://x.com/CryptoHodlerrrr/status/1688702767009800192`
[2] `https://x.com/muesliswapteam/status/1688546719225589760`

We reiterate the BPS equation, as presented in 5.8, adjusting them to the batcher model:

$$u_{MM}(B) = v_{MM}(B) + \sum_{t \in B} f_t - q$$

In the MuesliSwap protocol there is no burning mechanism, and the fees are constant. Since there are multiple pure revenue opportunities to be exploited by a monopolistic [3] matchmaker, it is logical to assume that his valuation is additive. Hence, $v_{MM}(B) = \sum_{t \in B} \mu_t$. Thus,

$$u_{MM}(B) = \sum_{t \in B} \mu_t + m \cdot f, \text{where m is the number of the aggregated transactions}$$

The maximization of $\sum_{t \in B} f_t$ is a variation of the Knapsack Extractable Value. The number of aggregated transactions is upper bounded by the number of inputs and outputs a eUTxO transaction can have, which in the current iteration of Cardano is 64. Also, in the correct MuesliSwap iteration the fees per swap is 1.7 ADA. Hence transaction fees are relatively low to the profit of a pure revenue opportunity exploitation, and we will no focus of them.

Suppose that, without loss of generality, a buy order $O = (X, Y, n, p', f)$, where $p' \in (p, \frac{100p+s}{100})$, gets executed instantly. Thus, $a' < p'$. Since we are interested in profit maximization by the matchmaker, let $p' = \frac{100p+s}{100}$. We distinguish between two different matchmaker strategies s_d and s_r, depending on the matchmaker's rationality model.

- s_h: If the matchmaker were *dummy* and tried to maximize the traders' utility, they would execute the swap at a price within the range (a', p'). Hence, $\mu = 0$. Note that this outcome is desirable for the traders because the swap price is within their preferred price range.

- s_r If the matchmaker were *rational*, they would execute the following strategy: They would set different prices for the buyer and the seller, specifically their maximum slippage prices. Given that n tokens are swapped in this trade, their pure revenue would be $\mu_O = n \cdot (p' - a')$. This represents the maximum payoff they can achieve from this trade.

We can see that $v_{MM}(s_r) = n \cdot (p' - a') > 0 = v_{MM}(s_h)$ Thus, s_d is strictly dominated by s_r, and a rational matchmaker will always choose to behave rational when given the chance.

Of course, an argument can be made that rational behavior can damage the DEX's credibility, acting as a counter incentive for this kind of behavior. We will avoid assuming anything about the DEX's long term strategic goals, and will instead focus on the DEX's roll as a matchmaker.

[3]In fact there were only two distinct matchmaking entities on the MuesliSwap protocol at the time of the exploit, both approved by the MuesliSwap team: https://cexplorer.io/asset/asset1urgqlyjezmj2cy50x28zm3zpl4n78y99cjl3ck/owner#data

Figure 6.3: A visualization of the rational matchmaker's strategy. The length of the $a'p'$ interval indicates the pure revenue of the matchmaker per token swapped.

Observe that the rational execution of this trade does, in fact, validate the transaction validator scripts guarding the traders' UTxOs.

We note that our goal is not to determine the optimal strategy for the matchmaker but simply to prove that the activity observed by the Cardano community can indeed be classified as an instance of MEV.

Notice that this strategy can be applied to every instantaneously matched order entering the LOB. Hence, a good, though perhaps not optimal, choice for the adversary is to order incoming transactions based on the pure revenues their individual exploitation provides and to execute them in decreasing order. Finding the optimal strategy for the adversary is beyond the scope of this work.

6.3 MinSwap

MinSwap [47] is the most dominant Cardano DEX in terms of Total Value Locked (**TVL**)[4] as of September 2024. MinSwap's original design involved developing a CPMM, using smart contracts that users could interact with on-chain. However, this design could not achieve the desired transaction throughput due to concurrency issues [48]. As explained in 3.3.2, due to the DEX's high traffic, multiple users were attempting to reference a specific smart contract UTxO in the same block. This resulted in only a single transaction being processed while all others were reverted.

To address concurrency, the MinSwap team opted for off-chain transaction execution through a transaction batcher protocol named Laminar [49]. At the time of writing, the only entity allowed to operate a Laminar batcher is MinSwap Labs themselves. Therefore, Laminar operates under a trusted third-party ordering scheme as described in [39].

Users can interact with MinSwap's DeFi smart contracts to provide additional liquidity or swap tokens in one of its pools. To do so, they must submit their transactions along with the corresponding funds to MinSwap's batcher smart contracts. Each transaction carries metadata associated with the user, including their financial intent (redeemer, desired price, timestamp). Like all batcher aggregation methods, the submitted transaction will only be executed if the appropriate conditions are met, i.e., if the desired swap price is reached.

At the end of each Ouroboros round, the batcher will gather all the UTxOs locked

[4]TVL is a common term in DeFi, referring to the total monetary value locked in a protocol's smart contracts. You can view the Cardano projects' TVL here

in the batching smart contract. Then, these transactions will be ordered by their corresponding hashes, as they appear in the batcher smart contract. Transaction ordering initially occurred in ascending order until an adversarial entity orchestrated a series of sandwich attacks by exploiting this fact. We will explore this attack shortly. The output of this off-chain aggregation is then published on the blockchain, thus finalizing the transactions.

6.3.1 Sandwich attacks in MinSwap

In MinSwap's incident report [50], it was revealed that an attacker exploited the batching algorithm to carry out a sandwich attack on several trades occurring on the DEX. According to the team's report, the attack drained approximately 11k ADA from MinSwap's smart contracts, successfully sandwiching around 880 trades.

The attacker's strategy was as follows: they monitored the batcher smart contract for UTxOs of large transactions that could cause sufficient slippage in the CPMM, making the attack profitable. When another user submitted a qualifying transaction, the attacker observed its hash and created a front-running transaction with a smaller hash. Since cryptographic hash function outputs follow a uniform distribution over the function's codomain [51], this process is computationally feasible in the average case. The attacker repeatedly adjusted the parameters of their order to change the transaction's hash. Because hash generation is not computationally difficult, the attacker could continue until finding a transaction with a sufficiently low hash to front-run the victim's transaction. Such attacks, often referred to as "nothing at stake" [52].

To complete the sandwich attack, the adversary only needs to back-run the victim's transaction. This is easier than executing the front-running transaction. The adversary simply needs to ensure that their transaction is included in the same batch as the victim’s transaction but after it. Alternatively, the adversary can submit the back-running transaction in any future batch.

The aforementioned attack was executed by exploiting the logic within MinSwap's smart contracts. It was an atomic and risk-free attack, thereby compromising the DEX's technical security as defined in [21].

Below, we will do a very basic calculation about the probability of the above-mentioned attack's success.

Let Y be the hash of the victim's transaction, normalized over the co-domain of the hash function. Also, let X be the hash of the front-running transaction, normalized likewise. We assume that Y and X are independent and identically distributed (iid) from the Uniform $(0, 1)$ distribution. We calculate that the front-running attack has a $\frac{1}{2}$ probability of succeeding on the first try.

$$P(X < Y) = \int_0^1 \int_0^y f_X(x) f_Y(y)\, dx\, dy = \int_0^1 \int_0^y 1 \cdot 1\, dx\, dy \Rightarrow$$

$$P(X < Y) = \int_0^1 \left(\int_0^y 1\, dx \right) dy \Rightarrow$$

$$P(X < Y) = \int_0^1 y\,dy = \left[\frac{y^2}{2}\right]_0^1 = \frac{1}{2} \Rightarrow$$

$$P(X < Y) = \frac{1}{2}$$

Even for an arbitrary constant $Y \in (0, 1)$, the number of attempts, indicated as X, until a front-running hash is found, follows the geometric distribution:

$$P(X = k) = (1 - Y)^{k-1}Y$$

Where:

- Y is the probability of success on each trial,
- X is the number of trials until the first success,
- k is the trial on which the first success occurs.

We see that the number of attempts until the first success is $O(\frac{1}{2}^k)$

Also, the expected number of attempts until success is $E[X] = \frac{1}{Y}$, which scales linearly as Y approaches 0.

We note that since hashing is computationally (and thus financially) inexpensive, the only limiting parameters for the adversary are the time available to execute the aforementioned attack and the hash of the victim transaction. The front-running transaction must be included in the same batch as the victim transaction, meaning the available time window lies between when the transaction is added to the batching smart contract and when the batcher collects all transactions for aggregation. This topic is outside the scope of this work, so we digress.

6.4 SundaeSwap

SundaeSwap[53] was the first CPMM DEX to launch on Cardano; unfortunately, it fell victim to significant concurrency issues right at launch[5]. As a result, SundaeSwap had to address these concurrency issues, introducing a batching solution similar to those implemented by the previous two DEXs.

As mentioned earlier, SundaeSwap operates under the CPMM model. The primary difference from MinSwap's model is that SundaeSwap has decentralized its batching scheme, introducing batchers who are not directly affiliated with the SundaeSwap team. To mitigate MEV instances, they have selected a limited number of Cardano BPs and provided them with a batching license valid for one week. Consequently, SundaeSwap's batching ecosystem falls under the category of Professional Market Makers, as outlined in [39].

At the end of this period, only if the batcher "is in good standing with the SundaeSwap team" (which we assume it entails that the batcher has not exploited any MEV

[5] https://cointelegraph.com/news/sundaeswap-launches-on-cardano-but-users-report-failing-transactions

opportunities), will the batcher be able to collect their accumulated batching fees.

This arrangement is interesting because it is more decentralized than MinSwap's, while also imposing tangible penalties for exploitative behavior. This concept resembles slashing[54] in Ethereum, where BPs that fail to participate in the consensus mechanism as required face monetary penalties. The intricacies of the Ethereum consensus mechanism are not the focus of this work, so we digress.

We note that even in this context, the MEV definition provided by Roughgarden and reiterated in subsection 5.8 remains applicable due to the q parameter that accounts for burned funds.

In each round of the protocol, SundaeSwap market makers create aggregate transactions that reference the traders' UTxOs locked in the batching smart contract and the CPMM's liquidity pool UTxO. The outputs of this batch transaction must satisfy the traders' specified economic intent (i.e., the swap price must be within their accepted price range).

In subsection 3.3.2, we specified that if multiple eUTxO transactions referencing the same UTxO are included in the same block, all but the first one (according to their block ordering) will fail. Therefore, only a single batch transaction can be included in the block per round.

This property of the eUTxO model forces matchmakers into a race. The batcher who manages to submit their batch transaction first to the mempool of the BP responsible for block production in that specific round[6] is the one who wins the race, and thus reaps the batching rewards for that specific round.

6.4.1 Fatal front-running SundaeSwap batch transactions

What we explained above is an execution of the SundaeSwap protocol as intended by the designers, i.e., all the entities are described by the dummy model. Below, we will describe ways that rational entities can diverge from the intended execution of the protocol to increase their utility functions.

Let's first identify the actors of this protocol (i.e. the players), for a given round of Cardano protocol and give their utilities when playing strategy s_d, i.e. when behaving in a dummy manner:

- The matchmakers $M_1, \ldots, M_k$. We will use the single-minded value definition of 5.8. $u_{M_i}(s_d) \begin{cases} 1 & \text{if i gets their batch in the block,} \\ 0 & \text{otherwise.} \end{cases}$
- The block producer BP, $u_{BP}(s_d) = \sum_{t \in B} f_t$
- The relay node operators, $R_1, \ldots, R_l$. Since the relay nodes do not get any monetary incentives in Cardano, $u_{R_i}(s_d) = 0, \forall i \in (1, \ldots, l)$

[6] The BPs for each round are known ahead of time, as stated in [18]

Note that in the Cardano network, a BP controls a block-producing node along with two relay nodes that are neighboring the block-producing node in the network graph. For convenience, we will contract these two relay nodes along with the block-producing node in the network graph and treat them all as a single vertex. We will also assume that the BP does not control any other relay nodes in the network graph. This is not a strong assumption, because even if they did control more relay nodes, we can treat their utility function as the sum of the respective utilities.

Additionally, since every matchmaker is one of the Cardano BPs, there will be rounds where the BP for the round is one of the matchmakers. Hence, if $\exists i : M_i = BP \Rightarrow u_{BP}(s_d) = \sum_{t \in B} f_t + 1$ and $u_{M_{j \neq i}}(s_d) = 0$.

Suppose that when all matchmakers behave in a dummy fashion, then one of them is picked uniformly at random and has their batch transaction included in the block. In this case

$$E[u_{M_I}(s_d)] = \frac{1}{k} \cdot 1 + \frac{k-1}{k} \cdot 0 = \frac{1}{k}$$

A matchmaker can utilize off-chain agreements 5.9 with that round's BP, tipping the BP τ so that the BP prioritizes their transaction over the rest of the matchmakers.

Suppose now that M_i chooses to utilize OCA with the BP, and is the only matchmaker to do so. We name this strategy s_{OCA} and the tip will be a part of M_i's batching fees, hence $\tau \in (0, 1)$. Since

$$u_{BP}(s_{OCA}) = \sum_{t \in B} f_t + \tau > \sum_{t \in B} f_t = u_{BP}(s_d)$$

A rational BP would accept this offer. Thus, $u_{M_i} = 1 - \tau$. We notice that for $\tau \in (0, \frac{k-1}{k})$

$$\tau < \frac{k-1}{k} \Rightarrow \tau < 1 - \frac{1}{k} \Rightarrow t - 1 < -\frac{1}{k} \Rightarrow 1 - \tau > \frac{1}{k}$$

$$\text{Thus, } u_{M_i}(s_{OCA}) > E[u_{M_i}]$$

Hence, given that M_i is the only one utilizing off-chain agreements, on expectation, s_{OCA} strictly dominates u_{M_i}. This is highly problematic for SundaeSwap's protocol.

Also, in this scenario, only one matchmaker is rational. Many players might attempt to utilize off-chain agreements. Additionally, since the bidding data is off-chain, no matchmaker can know the bids of the others, thus creating a sealed-bid auction [55] with the BP as the auctioneer.

Furthermore, since there is a significant number of possible Block Producers in the Cardano network[7], some matchmakers might choose to tip relay nodes to influence them. If a single matchmaker or a coalition of matchmakers could influence a set of relay nodes that forms a dominating set in Cardano's network graph, they could prioritize their transactions over those of their peers, thereby reaping all of the batching fees.

[7]At the time of writing, this number is 2,763

Of course, the fatal front-running does not occur within the SundaeSwap protocol itself; rather, it affects SundaeSwap batch transactions and takes place on the Cardano network. This highlights how the large total value locked (TVL) in a DEX that operates on top of a blockchain can impact the blockchain itself. This phenomenon is already known in Ethereum and relates to how significant on-chain tipping or off-chain agreements can overshadow a protocol's monetary rewards to BPs, thereby altering rational BPs' behavior. This, in turn, threatens the protocol's safety at the consensus level, as noted in [40].

CHAPTER 7

CONCLUSION

After investigating the three main Cardano DEXs, along with some of the security vulnerabilities present in them, it is now clear that the inner workings and limitations of the eUTxO accounting model greatly affect Cardano DeFi. The concurrency issue has led all the DEXs we analyzed to adopt a batching solution as a workaround. Alas, batching smart contracts introduce a single point of failure for the DeFi infrastructure, making them a significant target for any rational entity wishing to maximize profits. Additionally, their permissioned nature undermines the decentralization of the Cardano protocol and gives the DEXs strong monopolistic positions in the DeFi landscape.

In the MuesliSwap incident, we observed that the DEX team was able to leverage their monopolistic position as the sole matchmakers to extract additional profits from protocol users.

In the Minswap sandwich attack, an adversary exploited the batching algorithm of the DEX to execute several sandwich attacks and extract profits from traders, with no cost or risk to themselves. Although the specific exploit has since been patched, the existence of a single batching smart contract remains a point of failure for the entire DEX, making it a significant vulnerability for future adversaries.

Finally, in the case of SundaeSwap, there have not been any reported instances of MEV targeting traders. Nevertheless, the batching completion introduced by the DEX's decentralized model creates strong monetary incentives for BPs, relay nodes, and matchmakers to establish off-chain agreements for transaction prioritization. This could lead to a shadow economy on the Cardano consensus layer, impacting the overall decentralization and security of the protocol.

7.1 Future Work

We believe that a drastic rework of the eUTxO transaction aggregation is needed to address the aforementioned limitations. There are two routes one can explore: either a rework of the batch ordering system or a complete abolishment of it in favor of novel

primitives.

The main problem with the batching scheme is its lack of decentralization on both the batching and liquidity sides. On the batching side, batchers have total control over transaction ordering, incentivizing them to engage in MEV practices. We believe that enhancing the batching scheme with additional ordering limitations could help mitigate this issue. There are several routes one could take, as presented in [39]:

- **Professional matchmaking**: Currently, the monetary incentives for matchmakers are rather lackluster. We believe that allowing traders to tip their matchmakers as they see fit, coupled with a slashing mechanism similar to that of SundaeSwap, could bring transparency to the process and deter many MEV exploits.
- **Algorithmic Committee ordering**: Another route is to introduce additional ordering rules regarding the time of inclusion of each transaction in the batching contract. Since time in the blockchain can be quite variable, one could implement a fair-ordering scheme like the one presented in [56] by Kiayas, Leonardos, and Shen.
- **Commit and reveal schemes**: The final avenue we believe should be explored in this direction is the inclusion of Zero-Knowledge (ZK) proofs [57]. ZK proofs can be utilized to mask the depth and price of transactions, thereby obscuring their impact on the market. This results in matchmakers not knowing the depth of a transaction before execution, making it difficult for them to identify pure revenue opportunities.

Another direction is to abolish batching schemes altogether. A novel idea presented in [58] suggests encoding unexecuted trades as internal transactions within an eUTxO account, allowing DEX-agnostic matchmakers to process them as long as they fulfill the economic intent set by the trader. There is currently no formal analysis of this idea, nor of the encoding process of a DeFi construction that utilizes it.

BIBLIOGRAPHY

[1] S. Nakamoto, ``Bitcoin: A peer-to-peer electronic cash system," 2008.

[2] R. L. Rivest, A. Shamir, and D. A. Wagner, ``Time-lock puzzles and timed-release crypto," tech. rep., USA, 1996.

[3] J. A. Garay, A. Kiayias, and N. Leonardos, ``The bitcoin backbone protocol: Analysis and applications," *Journal of the ACM*, 2015.

[4] V. Buterin *et al.*, ``Ethereum white paper," *GitHub repository*, vol. 1, pp. 22--23, 2013.

[5] V. Buterin, ``The x*y=k market maker model." Online, 2018. https://ethresear.ch/t/improving-front-running-resistance-of-x-y-k-market-makers/.

[6] H. Adams, N. Zinsmeister, M. Salem, R. Keefer, and D. Robinson, ``Uniswap v3 core," *Tech. rep., Uniswap, Tech. Rep.*, 2021.

[7] C. HOSKINSON, ``Cardano whitepaper," 2017. Accessed: September 13, 2024.

[8] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.

[9] M. J. Osborne, ``An introduction to game theory," *Oxford University Press google schola*, vol. 2, pp. 672--713, 2004.

[10] L. Lamport, ``Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51--58, 2001.

[11] M. Castro, B. Liskov, *et al.*, ``Practical byzantine fault tolerance," in *OsDI*, vol. 99, pp. 173--186, 1999.

[12] I. Eyal and E. G. Sirer, ``Majority is not enough," *Communications of the ACM*, vol. 61, pp. 95 -- 102, 2013.

[13] M. Weinberg and A. Narayanan, ``On the instability of bitcoin without the block reward," *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[14] IOHK, ``Dynamic p2p is coming to cardano," 2023. Accessed: 2024-09-18.

[15] C. Foundation, ``Cardano p2p networking," 2024. Accessed: 2024-09-18.

[16] A.-M. Kermarrec and M. Van Steen, ``Gossiping in distributed systems," *ACM SIGOPS operating systems review*, vol. 41, no. 5, pp. 2--7, 2007.

[17] B. M. David, P. Gazi, A. Kiayias, and A. Russell, ``Ouroboros praos: An adaptively-secure, semi-synchronous proof-of-stake blockchain," in *International Conference on the Theory and Application of Cryptographic Techniques*, 2018.

[18] A. Kiayias, A. Russell, B. David, and R. Oliynykov, ``Ouroboros: A provably secure proof-of-stake blockchain protocol," in *Annual international cryptology conference*, pp. 357--388, Springer, 2017.

[19] M. Miller, ``The future of law," *paper delivered at the Extro*, vol. 3, 1997.

[20] N. Szabo, ``Formalizing and securing relationships on public networks," *First monday*, 1997.

[21] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, and W. Knottenbelt, ``Sok: Decentralized finance (defi)," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, AFT '22, (New York, NY, USA), p. 30–46, Association for Computing Machinery, 2023.

[22] M. M. T. Chakravarty, J. Chapman, K. M. Mackenzie, O. Melkonian, M. P. Jones, and P. Wadler, ``The extended utxo model," in *Financial Cryptography Workshops*, 2020.

[23] G. Wood *et al.*, ``Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1--32, 2014.

[24] J. E. Hopcroft, R. Motwani, and J. D. Ullman, ``Introduction to automata theory, languages, and computation," *Acm Sigact News*, vol. 32, no. 1, pp. 60--65, 2001.

[25] V. Buterin and V. Griffith, ``Casper the friendly finality gadget," *arXiv preprint arXiv:1710.09437*, 2017.

[26] J. Zahnentferner, ``Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies," *Cryptology ePrint Archive*, 2018.

[27] D. Zindros, ``Blockchain foundations." https://ee374.stanford.edu/blockchain-foundations.pdf#page=41.47. Accessed: 2024-08-23.

[28] J. Zahnentferner, ``An abstract model of UTxO-based cryptocurrencies with scripts." Cryptology ePrint Archive, Paper 2018/469, 2018. https://eprint.iacr.org/2018/469.

[29] A. Chepurnoy, V. Kharin, and D. Meshkov, ``Self-reproducing coins as universal turing machine," in *International Workshop on Data Privacy Management*, pp. 57--64, Springer, 2018.

[30] A. M. Turing *et al.*, ``On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[31] L. Brünjes and M. J. Gabbay, ``Utxo- vs account-based smart contract blockchain programming paradigms," *ArXiv*, vol. abs/2003.14271, 2020.

[32] M. D. Gould, M. A. Porter, S. Williams, M. McDonald, D. J. Fenn, and S. D. Howison, ``Limit order books," *Quantitative Finance*, vol. 13, no. 11, pp. 1709--1742, 2013.

[33] J. Xu, K. Paruch, S. Cousaert, and Y. Feng, ``Sok: Decentralized exchanges (dex) with automated market maker (amm) protocols," *ACM Computing Surveys*, vol. 55, no. 11, pp. 1--50, 2023.

[34] Y. Zhang, X. Chen, and D. Park, ``Formal specification of constant product (xy= k) market maker model and implementation," *White paper*, 2018.

[35] L. Zhou, K. Qin, C. F. Torres, D. V. Le, and A. Gervais, ``High-frequency trading on decentralized on-chain exchanges," in *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 428--445, IEEE, 2021.

[36] Y. Wang, ``Automated market makers for decentralized finance (defi)," *arXiv preprint arXiv:2009.01676*, 2020.

[37] J. Milionis, C. C. Moallemi, and T. Roughgarden, ``Complexity-approximation trade-offs in exchange mechanisms: Amms vs. lobs," in *International Conference on Financial Cryptography and Data Security*, pp. 326--343, Springer, 2023.

[38] B. Mazorra, M. Reynolds, and V. Daza, ``Price of mev: Towards a game theoretical approach to mev," in *Proceedings of the 2022 ACM CCS Workshop on Decentralized Finance and Security*, DeFi'22, (New York, NY, USA), p. 15–22, Association for Computing Machinery, 2022.

[39] L. Heimbach and R. Wattenhofer, ``Sok: Preventing transaction reordering manipulations in decentralized finance," *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, 2022.

[40] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, and A. Juels, ``Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability," in *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 910--927, 2020.

[41] V. Buterin, ``The searcher-builder-proposer separation and the anti-censorship properties of ethereum," 2024. Accessed on May 11, 2024.

[42] M. Bahrani, P. Garimidi, and T. Roughgarden, ``Transaction fee mechanism design with active block producers," *ArXiv*, vol. abs/2307.01686, 2023.

[43] T. Roughgarden, ``Transaction fee mechanism design," *J. ACM*, vol. 71, aug 2024.

[44] cardanians.io, ``Understanding cardano batchers," 2024. Accessed: September 13, 2024.

[45] M. Labs, ``Musliswap whitepaper," 2021. Accessed: 2024-06-12.

[46] MuesliSwap Team, ``Matchmakers - muesliswap documentation," 2022. Accessed: 2024-09-15.

[47] L. Nguyen, ``Minswap whitepaper," 2021. https://docs.minswap.org/get-started/whitepaper.

[48] MinSwap, ``Minswap testnet reflections," 2022. https://medium.com/minswap/minswap-testnet-reflections-64b01c5e7c45.

[49] MinSwap, ``Introducing laminar: An eutxo scaling protocol for accounting-style smart contracts," 2022. https://medium.com/minswap/introducing-laminar-an-eutxo-scaling-protocol-for-accounting-style-smart-contract-d1ac8847dde8.

[50] MinSwap, ``Incident report: Sandwich attack patch," 2022.

[51] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007.

[52] E. Deirmentzoglou, G. Papakyriakopoulos, and C. Patsakis, ``A survey on long-range attacks for proof of stake protocols," *IEEE access*, vol. 7, pp. 28712--28725, 2019.

[53] S. Labs, ``Sundaeswap whitepaper," 2022. Accessed: 2024-06-12.

[54] Z. He, J. Li, and Z. Wu, ``Don't trust, verify: The case of slashing from a popular ethereum explorer," in *Companion Proceedings of the ACM Web Conference 2023*, pp. 1078--1084, 2023.

[55] T. Roughgarden, ``Algorithmic game theory," *Communications of the ACM*, vol. 53, no. 7, pp. 78--86, 2010.

[56] A. Kiayias, N. Leonardos, and Y. Shen, ``Ordering transactions with bounded unfairness: Definitions, complexity and constructions." Cryptology ePrint Archive, Paper 2023/1253, 2023. https://eprint.iacr.org/2023/1253.

[57] A. Nitulescu, N. Paslis, and C. Rafols, ``Flip -and-prove r1cs,"

[58] I. Oskin, ``Spectrum bloom: A self-developing, sustainable, eutxo-native framework for decentralized finance," 2023.