TU WIEN Informatics

Analysis of Front Running Vulnerabilities in Solidity Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Halid Zećirović, BSc. BSc.
Matrikelnummer 01350673

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ass.Prof.in Dr.in Monika di Angelo
Mitwirkung: Ao.Univ.Prof. Dr. Gernot Salzer

Wien, 8. Oktober 2024

Halid Zećirović	Monika di Angelo

TU WIEN Informatics

Analysis of Front Running Vulnerabilities in Solidity Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Halid Zećirović, BSc. BSc.
Registration Number 01350673

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof.in Dr.in Monika di Angelo
Assistance: Ao.Univ.Prof. Dr. Gernot Salzer

Vienna, October 8, 2024

Halid Zećirović	Monika di Angelo

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Halid Zećirović, BSc. BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, habe ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT-Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 8. Oktober 2024

Halid Zećirović

Danksagung

An erster Stelle möchte ich mich bei Monika di Angelo und Gernot Salzer bedanken, die mir durch ihre herausragende Betreuung den Weg geebnet haben, diese Arbeit fertigzustellen.

Ein besonderer Dank gilt meinen Eltern, deren harte Arbeit dieses Studium erst ermöglichte, sowie meinen Schwestern, die mich stets ermutigten, diesen Weg beizubehalten. Ebenso danke ich meiner Frau für ihren täglichen Beistand über all die Jahre.

Nicht zuletzt bedanke ich mich bei allen Weggefährten, die mit mir unzählige Stunden mit Projekten, Abgaben und Beispielen geteilt haben.

Kurzfassung

Blockchain-Technologien, insbesondere Smart Contracts auf Plattformen wie Ethereum, haben in den letzten Jahren sowohl bei Entwicklerinnen und Entwicklern als auch bei Forscherinnen und Forschern stark an Bedeutung gewonnen. Dennoch gibt es in diesem schnell wachsenden Ökosystem immer wieder Sicherheitsprobleme, die des Öfteren für Schlagzeilen in den Medien sorgen. Das Hauptaugenmerk dieser Arbeit liegt auf einer bestimmten Kategorie von Schwachstellen, dem sogenannten Front-Running. Diese Schwachstelle nutzt eines der Grundprinzipien der Blockchain-Technologie, nämlich ihre Transparenz, aus, um sich Vorteile zu verschaffen und den Opfern erhebliche finanzielle Verluste zuzufügen. Ziel dieser Arbeit ist es, eine gründliche Analyse dieser Art von Schwachstellen in Solidity-Smart-Contracts durchzuführen.

Zunächst werden grundlegende Informationen zur Blockchain-Technologie und zu Smart Contracts gegeben, wobei Ethereum und dessen dazugehörige Programmiersprache Solidity im Mittelpunkt stehen. Außerdem wird eine Sammlung von dokumentierten Solidity-Codebeispielen präsentiert, die verschiedene Angriffsszenarien für bekannte Unterkategorien demonstrieren. Nach einer ausführlichen Beschreibung von Front-Running und dessen Auswirkungen in dezentralisierten Programmen wird ein Überblick über die bestehende Forschung zu diesem Thema gegeben. Mittels einer gründlichen Literaturrecherche werden unterschiedliche Definitionen betrachtet und analysiert, um ein umfassendes Verständnis von Front-Running zu schaffen.

Um auf die aktuelle Situation einzugehen, wird das Phänomen des Maximum Extractable Value (MEV) genauer analysiert, wobei der historische Hintergrund, seine Auswirkungen und schlussendlich auch einige Fallstudien realer Angriffe und deren Folgen betrachtet werden.

Abstract

Blockchain technologies, especially smart contracts on platforms such as Ethereum, have gained significant popularity in recent years among developers and researchers. However, security concerns continue to plague this ecosystem, which provides a consistent stream of headlines to media outlets. The main focus of this thesis is a specific class of vulnerabilities known as front-running. This vulnerability exploits one of the fundamental principles of blockchain technology, its transparency, to gain unfair advantages and inflict significant financial losses on victims. This thesis aims to provide a thorough analysis of front-running vulnerabilities in Solidity smart contracts.

We begin by providing a comprehensive background on blockchain technology and smart contracts, with Ethereum and its programming language, Solidity, as the primary focus. We also present a collection of documented Solidity code examples demonstrating different attack scenarios for all known front-running subcategories. After thoroughly describing the concept of front-running in decentralized applications, we provide a comprehensive overview of existing research on the topic. By means of rigorous literature research, different definitions will be analyzed to provide a well-rounded understanding of front-running.

To address the current situation, we analyze the phenomenon of Maximum Extractable Value (MEV) in more detail, analyzing its historical background, its real-world impact, and finally presenting case studies of real attacks and their aftermath serving as a demonstration of the practical relevance and applicability of our research in understanding and mitigating front-running vulnerabilities.

Contents

CHAPTER 1

Introduction

Since its inception in 2015, Ethereum has been envisioned by its founders as a platform for building a wide array of different applications rather than another highly specialized version of a "cryptographically secure, transaction-based state machine" that they saw in Bitcoin [1]. Therefore, the Ethereum project is the attempt to build the generalized technology on which "all transaction-based state machines can be built". Over the years, Ethereum has evolved into an ecosystem of considerable size and importance, with its original vision being realized through the Ethereum Virtual Machine (EVM), an emulated decentralized computer, often referred to as the *world computer* in marketing materials.

Bitcoin, to this day the industry's figurehead, was introduced in 2008 by an anonymous entity under the pseudonym *Satoshi Nakamoto* and is widely credited as a foundational pillar of the blockchain industry [2]. Nonetheless, the origins of blockchain technology can be traced back to a 1982 dissertation authored by David Chaum, who later went on to create the first digital currency called *E-Cash* [3].

The blockchain ecosystem has evolved significantly from there, with many projects emerging to address diverse sets of challenges. A significant advancement in this area is the possibility for developers to interact with the blockchain through scripts or entire computer programs. In the context of Ethereum, these programs are commonly referred to as *Smart Contracts*, which are decentralized pieces of software running on the EVM.

As is the case with conventional software, they can contain hard-to-detect weaknesses or bugs. However, a key difference lies in the challenges related to modifications of Smart Contracts after deployment. Modifying a deployed Smart Contract typically requires complex procedures such as proxy patterns. As fixes after deployment can be difficult and costly, and Smart Contracts possibly manage large amounts of assets, it is crucial to ensure security during the development.

The Ethereum platform manages billions of dollars in transactions and assets, hosting a steadily increasing number of Decentralized Applications (dApps). The vast asset values

involved in these transactions and the lack of a central authority to oversee them make the platform a desirable target for malicious actors. Tarnished by continuous breaches resulting in $3.1bn of lost funds in 2022 alone, the so-called Decentralized Finance (DeFi) sector is far from achieving a level of trust that the conventional financial sector has had for decades [4].

Researching smart contract vulnerabilities is essential. Given that the Ethereum platform has been around for less than a decade, is proliferating, and is continuously evolving, research in this area is challenging to conduct and keep up with. This thesis aims to contribute to this field by formally investigating and specifying a subset of vulnerabilities: **Front Running (FR)**.

1.1 Problem Statement

Transactions on any blockchain, including Ethereum, are visible in the so-called *mempool*, a pool of unconfirmed transactions visible to all network participants before they are committed to the blockchain. This transparency allows a malicious observer to act in specific ways before the transaction is included in a block [5]. A classic example of this can be found in decentralized auctions, which allow attackers to *"front-run"* existing bids by increasing their bid's fee, ensuring that miners process their transaction first. Front-running can thus be described as

> *" [...] the race to order the chaos to the winner's benefit."* [5]

This work aims to thoroughly investigate the phenomenon of front-running in Ethereum and analyze its impact on the ecosystem. First, we formally define front-running by analyzing the different definitions and characteristics found in the literature.

After generating a clear and comprehensive definition, this work will give a detailed description of possible attack scenarios for front-running, analyze common characteristics of front-running, as well as study detection techniques and practical countermeasures.

A detailed taxonomy of front-running attack scenarios will be provided, which will be used to create a collection of documented code examples in Solidity for all known front-running subcategories and their corresponding exploits, which demonstrate different attack scenarios.

These contracts will then be used to investigate the behavior and limitations of existing detection tools and techniques. Furthermore, code patterns that are problematic for detection tools will be identified and documented.

Finally, recommendations for implementing mitigation techniques that can help prevent front-running in Smart Contracts will be provided.

The following research questions will be addressed:

1. **Definition** - What are the proper definitions, key characteristics, and mechanisms of front-running?

2. **Effects and Exploits** - How does front-running affect the security, efficiency, economics, and performance of smart contract ecosystems?

3. **Detection** - Which tools and methodologies are used to determine vulnerabilities?

4. **Mitigation** - What are techniques and strategies to mitigate front-running vulnerabilities, and how can they be implemented?

1.2 Methodology

The methodology can be divided into the following steps:

Literature Review

Firstly, a review of the latest literature will be conducted to document the present state of the research into Smart Contract vulnerabilities with a specific focus on front-running. A general overview of the impact of front-running on the security, efficiency, economics, and performance of smart contract ecosystems will be provided.

During the review, the most common definitions and characteristics of front-running will be identified, compared, and evaluated.

Finally, considering the aforementioned definitions, the theoretical foundations behind detection tools and their detection techniques will be examined.

Tool selection

Vulnerability detection tools and their detection techniques will be searched and selected according to clearly defined criteria (e.g., open source, recency, quality) and presented.

Example Contracts

A set of illustrative examples in Solidity will be provided for all known subcategories of front-running and their corresponding exploits.

Evaluation

Finally, the example contracts will be used to evaluate the tools and their detection techniques, providing insights into their behavior and limitations.

1.3 Structure

This thesis is structured as follows:

Chapter 2 provides a brief overview of blockchain technology and its terminology to give the reader the necessary context.

Front-running vulnerabilities, historical background and concrete examples are presented and discussed in chapter 3.

A systematic literature review is performed in chapter 4 to extract, discuss and evaluate definitions of front-running and related concepts.

The concept of Maximal Extractable Value (MEV) and its relation to front-running is elaborated in chapter 5 followed by a conclusion of the thesis in chapter 6 including a discussion of the results and outlooks on future work.

CHAPTER 2

Background

This chapter provides a brief overview of relevant technologies for this thesis. Technical implementations of decentralized blockchain systems are explicitly out of the scope of this work. Instead, we will focus on the theoretical background of blockchain technology and the concept of Smart Contracts.

2.1 Blockchain technology

The blockchain is a distributed data structure that is shared among the participants of a network [6]. Its primary function is to append *blocks*, which contain data in the form of a list of transactions 2.3, to a continuously growing chain of previous blocks with each block having a *hash-pointer* to the previous block. A hash-pointer is a cryptographic hash of the previous block's header, which is included in the current block's header. The inclusion of these hashes is a critical feature of the blockchain, as this mechanism ensures that the data structure is virtually tamper-proof and immutable.

In other words, once a block is added to the chain, it cannot be altered or deleted after more blocks have been appended. A blockchain can, therefore, also be referred to as a *"[...] very specialized version of a cryptographically secure, transaction-based state machine"* [7].

Adding new blocks to the blockchain involves a network of participants, usually referred to as *nodes*. These nodes work together to validate and record new transactions. *Consensus mechanisms* (section 2.1.2) are used to ensure that all nodes agree on one state of the blockchain and select valid transactions that can be included.

2.1.1 Forks

A *fork* is essentially a disagreement between nodes in the network regarding which chain is valid. Therefore, multiple states of the system coexist at approximately the same time,

with some nodes accepting one block as the next valid one while other nodes accept a block, which may contain radically different transactions, as the valid one instead [7].

Forks can also be applied intentionally, whereas one usually differentiates between *hard forks* and *soft forks*. A hard fork contains fundamental changes to the blockchain protocol or other software updates, whereby validation rules are changed. Participants who continue validation according to the old rules will no longer be able to produce valid blocks. A soft fork, on the other hand, contains updates that are backward compatible with existing rules. For any fork, the network may split into two separate chains that can coexist.

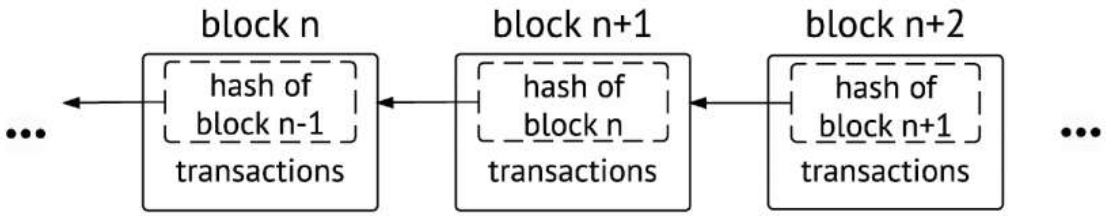

Figure 2.1: Illustration of a blockchain by Christidis and Devetsikiotis [6]

A blockchain contains several core features that are fundamental to its functioning. Sultan et al. [8] define four core aspects of blockchain technology, which we will briefly analyze.

Immutability

Due to the usage of cryptographic hashes for appending blocks, a blockchain is considered permanent and tamper-evident. More specifically, data cannot be modified once added to the chain. Although this creates a certain level of trust in the network [8], it also imposes several paradigm-changing implications on users.

This feature presents considerable challenges for Smart Contracts (see section 2.3) in particular. Although the inherent immutability of a blockchain ensures that the execution of the contract happens as agreed, fixing errors or bugs in the code is not as straightforward as in traditional software development.

Decentralization

In contrast to centralized systems that require some central authority to validate and permit transactions, a blockchain allows users to interact with each other without any middlemen. A decentralized network needs to be able to verify each block independently without relying on a central authority.

Consensus Driven

Different consensus mechanisms exist and are used on different blockchains. Consensus mechanisms must ensure that the network can continue to work even while assuming

that some malicious participants will exist.

Transparent

A blockchain is often referred to as a *decentralized public ledger of records*. It is *transparent* to the participants irrespective of whether it is public. If it is *public*, everyone can become a participant.

Even though blockchains have been described as anonymous [9], this is a common misconception of the technology. Permissionless blockchains usually allow users to create as many key pairs (which can be considered synonymous with accounts) as they wish without needing to provide any identifiable data.

Therefore, blockchains can be considered *pseudonymous* at best, as several de-anonymization techniques exist and have been used to identify criminals and other actors on the blockchain successfully [10] [11].

2.1.2 Consensus mechanisms

Consensus mechanisms are critical for solving the so-called *Byzantine Generals Problem* in decentralized networks [12]. This problem refers to the challenge of reaching consensus in distributed systems where some participants may act faulty or maliciously. Regarding blockchains, malicious nodes could attempt to destabilize the network by adding invalid transactions, such as trying to spend the same funds twice. Robust consensus mechanisms ensure that the network can withstand malicious actors to a certain degree.

Different consensus mechanisms exist for addressing these issues in blockchain systems, with two of them being the most popular:

- **Proof of Work (PoW)** - Nodes compete to solve a computationally intensive puzzle. The first node to solve this puzzle is allowed to add the next block.
- **Proof of Stake (PoS)** - Nodes are selected to validate transactions based on the cryptocurrency amount they stake. The more cryptocurrency a node holds, the higher the possibility of being chosen as a validator.

While Bitcoin still relies on PoW, Ethereum has transitioned to PoS in September 2022 [13]. This change also brings an important distinction regarding the entity responsible for choosing the next block. While in PoW, we refer to *miners*, in PoS, the term *validators* is used.

An important aspect of finding consensus is the concept of the *longest chain*. In Bitcoin, the longest chain is considered valid as it contains the most work. In Ethereum, this rule has been replaced by a *fork-choice algorithm* that measures the *weight* of the chain, which corresponds to the sum of validator votes, weighted by their stakes [14].

2.2 Ethereum

Ethereum is a blockchain platform first conceptualized in 2014 by Vitalik Buterin [15] and launched in 2015 [1]. The main difference to Bitcoin is that Ethereum aims to be a platform that can store data and execute code, thereby paving the way for a platform on which dApps can be built [1]. This can be achieved through smart contracts, which are small computer programs executed autonomously on the Ethereum network and written in the Turing complete programming language Solidity.

2.2.1 EVM

The execution of these programs is performed by the EVM, which is envisioned as a decentralized computer able to run arbitrary code [1]. The EVM is a stack-based virtual machine that executes bytecode instructions (*opcodes*).

Furthermore, the platform uses its proprietary cryptocurrency called *Ether*, which is used to pay for transaction fees and computation.

2.3 Terminology

Block

In general, a block is the central data structure of a blockchain. For Ethereum, a block consists of a *block header* and a list of transactions. The block header contains important information, including the previous block's hash, the block number, the transactions' root hash, and a timestamp [7, p.5-6].

Accounts

Each account in Ethereum consists of a 20-byte address and contains four fields:

1. **Nonce** - A counter that ensures that each transaction is executed only once.
2. **Ether balance**
3. **Contract code** (if it exists, for more information see section2.3)
4. **Storage**

Accounts are distinguished into two types: Externally Owned Account (EOA), controlled by their private keys, and Contract Account (CA), controlled by their contract code. An EOA has no code, incurs no cost for creation, and cannot create Smart Contracts. On the other hand, a CA has code executed after each message it receives; it can create other Smart Contracts and transfer Ether to other accounts.

State

In Ethereum, one often refers to the current *world state*, which is described as representing all information that can *"[...] currently be represented by a computer [...]"* and is a mapping between addresses and account states [1].

Smart Contracts

Smart Contracts are described as cryptographic *"boxes"* containing value that is unlocked if certain conditions are met [1]. It is essential to note that, although they are called contracts, they have little to do with traditional contracts and should be more accurately considered as autonomous agents living in the Ethereum execution environment [1]. More succinctly, smart contracts are small computer programs that are executed autonomously [16].

Transactions

Transactions are signed data packages containing a message that can be sent from any EOA [1]. In the Ethereum Yellowpaper, the driving motivation for the creation of Ethereum is *"[...] to facilitate transactions between consenting individuals who would otherwise have no means to trust one another"* [7, p.1] and Ethereum as a whole is described as *"transaction-based state machine"* [7, p.2]. This state, often formally referred to as σ, can be morphed into a new state σ' by means of transactions.

Therefore, transactions represent a *"[...] valid arc between two states"* [7, p.2]. The authors emphasize the importance of the term *valid* as there are many more invalid state changes than valid ones. A transaction, therefore, can be the transferring of Ether from one account to another, the creation of a new account, or the execution of a Smart Contract (see below). Any transaction contains the following fields:

- The recipient of the message
- The sender's signature
- The amount of Ether to be transferred to the recipient
- A data field (optional)
- A `startgas` value, which represents the maximum number of computational steps the transaction is allowed to take
- A **`gasprice`** value, which represents the fee the sender is willing to pay per computational step

Messages

The main difference between messages and transactions is that messages are created by CA and not EOA. CAs have the ability to send *messages* to other contracts. A message is produced whenever a contract executes the **`CALL`** opcode [1].

Gas

Fees on Ethereum are denominated in *gas* [7, p.13], which is the fundamental unit of network cost in Ethereum and used as a measure of computational effort. All transactions and contract executions consume gas, paid for in Ether. It is important to note that the gas price fluctuates depending on current network conditions [17]. The previously mentioned `gasprice` value is the maximum price the sender is willing to pay per gas unit.

If a transaction runs out of gas, all state changes are reverted except for the spent gas, thus helping to protect the network from being clogged up by infinite loops and compensating proposers who have executed the computational steps until the transaction ran out of gas [1].

The gas system is intended not only as an anti-DoS measure but also to ensure fairness regarding resource allocation. It furthermore allows for transaction prioritization as users can set a higher gas price to incentivize validators to include their transactions in the next block [1].

Mempool

The memory pool (usually abbreviated as *mempool*) is a set of pending transactions waiting to be confirmed. Miners (or validators) select transactions from this pool to include them in the next block they produce.

2.3.1 Decentralized Exchanges (DEXs)

A major use case of cryptocurrencies is trading. This is generally done in centralized exchanges, which act as intermediaries and often hold large amounts of user funds. This single point of failure has led to a significant number of hacks and subsequently to lost funds for users amounting to billions of dollars [18].

In the spirit of full decentralization, smart contracts have paved the way for the creation of DEXs, which allow users to trade cryptocurrency without any central authority. Although trading volume on centralized exchanges [19] exceeds that of decentralized exchanges [20] by far, the latter has seen significant adoption in recent years, with approximately 30% of trade volume coming from decentralized exchanges, with the most popular being Uniswap [21].

Classic exchanges usually employ an exchange design called *continuous-limit order book*, which consists of all open offers from buyers and sellers in the system. In this system, buyers place *bids* specifying at which price they are willing to buy an asset, and sellers place *asks* specifying at which price they are willing to sell an asset. The exchange then uses this information to match buyers and sellers [16].

In DEXs, these order books are managed using smart contracts with users holding their assets on the chain and the Smart Contract taking the role of exchange operators [16].

Different designs exist for DEXs, with the most popular one being the Automated Market Maker (AMM) model. Unlike traditional exchanges that use a continuous-limit order book model, AMMs operate through smart contracts that establish liquidity pools for token pairs. These pools are then used as reserves of assets and are manged by the contract itself.

2.3.2 Layer-2 solutions

Layer-2 solutions are technologies built on top of Ethereum's main chain (Layer 1), which are meant to improve transaction throughput and reduce gas costs ("scaling"). These solutions process transactions off the main chain while still leveraging its security. Various approaches exist for Layer-2 solutions, the most prominent one being rollups.

Rollups aggregate transactions, perform computations and storage off-chain, and submit the resulting transaction batches to the blockchain. A key component in rollup architectures is the *sequencer*. Sequencers are responsible for aggregating user transactions, executing them off-chain, and submitting the resulting transaction batches to the Layer-1 network. They are essential in ordering transactions and ensuring the timely processing of user operations within the Layer-2 ecosystem.

CHAPTER 3

State of the Art

Before the invention of Smart Contracts, front-running was already a mainstay in financial markets, where it had occupied regulatory authorities for decades. Standard economic research has long grappled with this phenomenon in securities exchanges and ways to mitigate it.

Historical accounts suggest that front-running likely first appeared on the Chicago Board Options Exchange (CBOE) in the 1970s, the first and most extensive options exchange in the world at the time [22]. This era saw explosive growth in liquidity and trading volume, leading to the invention of various forms of market abuse. Among those, one has been identified as front-running by the Securities and Exchange Commission (SEC) [22] and defined as:

> *The practice of trading a security while in possession of unreported information concerning a block transaction in the same or a related security.*

From this definition, one can see that in classical markets, front-running can be considered a form of insider trading, a much more broadly defined financial crime in which people with non-public knowledge enrich themselves by acting or entering positions before the public has any opportunity to do so. The difference here is the time horizon, which in the case of front-running is, in most cases, more restricted. This practice remains subject to legal scrutiny and regulatory measures aimed at preserving the integrity of financial markets.

Markham [22] continues to subdivide front-running into three distinct categories:

1. **"Tippee" trading** – Trading by third parties who are tipped on impending trades
2. **Self front-running** – The purchaser of the trade itself hedging against future price changes

3. **Trading ahead** – A broker with knowledge of an impending customer trade trades ahead of that order

The old concept of front-running has taken on another dimension in decentralized finance, given that Smart Contracts are transparent and deterministic.

Since all data on a Smart Contract is publicly visible, the notion of *insider trading* does not exist. In decentralized blockchain platforms like Ethereum, where transactions are visible before they are confirmed, malicious actors may exploit this to anticipate and execute trades that take advantage of forthcoming market movements, thus gaining unfair profits.

3.1 Front Running in Decentralized Applications

Front-running in the context of Smart Contracts is often defined as a race condition vulnerability **with financial damage to the victim** [23]. It is a form of attack where a malicious actor observes a transaction before it is confirmed and then submits a transaction that takes advantage of the information in the observed transaction, leading to several attack patterns, which are commonly referred to as displacement, suppression and insertion attacks [24], and which will be discussed later.

An important distinction must be made between front-running, insider trading, and arbitrage. Eskandari et al. [24] distinguish between front-running and insider trading since front-running requires a concrete transaction to which the attacker reacts. In contrast, insider trading is based on access to non-public, more general information. Arbitrage, conversely, means reacting *after* a transaction is confirmed or information has reached the public. Therefore, arbitrage is commonly seen as beneficial for market efficiency.

For front-running to occur, a so-called Transaction Order Dependence (TOD) must exist as a precondition. TOD can be defined as having two sequences of the same transactions that lead to different final states. Front-running specifically takes advantage of the protocol that validates and broadcasts transactions to the network. Every transaction that gets broadcast is first relayed around the network and then put into a valid block by a node. Front-running occurs when attackers observe a transaction before it is confirmed, submit a transaction that takes advantage of the learned information, and cause the node to incorporate the transactions in an order that benefits the attacker [23].

Ethereum transactions require a fee usually represented in *gas* (see section 2.3). Each transaction requires a minimum amount of gas, but beyond this, the amount of gas can be adjusted by the user. The price (`gasPrice`) that users pay for transactions per gas unit can influence how quickly validators will process a transaction and include it in their blocks. In almost all cases, a profit-maximizing block proposer will prioritize transactions with the highest gas fee due to limited block space. This process is usually referred to as *gas auction*. Therefore, it is possible for any user running a full node to observe the

pool of unconfirmed transactions and use this financial incentive to influence the order of transactions, causing victim transactions to be executed in different blockchain states than expected [25].

It is important to note that even though this vulnerability is enabled by a fundamental aspect of the decentralized protocol, front-running is not necessarily malicious and heavily depends on the contract's internal logic and possible mitigations [24]. Therefore, for front-running to occur, not only must there be a TOD in the contract but also a financial incentive for the attack to be profitable.

Front-running attacks are closely related to the well-understood concept of double-spending.

Double-spending is a fundamental issue in cryptocurrency systems where users spend the same currency unit more than once. The double spending attack is paying (off-chain) for goods or services and reversing the transaction after receiving them by exploiting the necessary time to confirm the transaction [26]. It can therefore be described as a form of *self-front-running* [24].

Eskandari et al. [24] describe three common templates under which front-running attacks can generally be categorized, depicted in fig. 3.1:

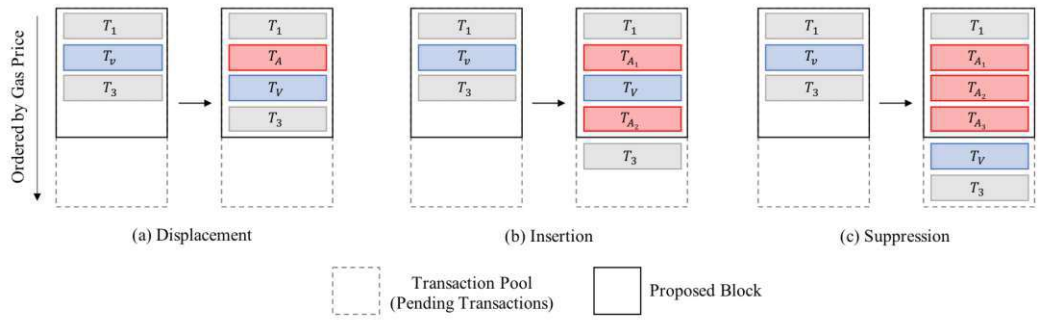

Figure 3.1: Illustration of common attack types by Torres et al. [27]

Displacement

> An attacker A observes a profitable transaction T_V of victim V and broadcasts their transaction to the network with a higher gas price so that block proposers will include T_A before T_V [25].

A displacement attack is usually launched when an attacker spots a potentially lucrative transaction in the mempool. The attack consists of the attacker sending their transaction with a higher gas price to incentivize proposers to prioritize their transaction over the initial one (therefore displacing it). This type of attack is used in various scenarios, such

as puzzle games where submitting the correct answer first wins a prize. If an attacker observes a submitted solution, they can create a similar transaction with a higher gas price, displacing the original transaction and claiming the reward.

Listing 3.1: Contract susceptible to displacement

```
pragma solidity ^0.8.0;

contract Displace {
    bytes32 puzzleHash;

    constructor(bytes32 _puzzleHash) payable {
        require(msg.value == 1 ether);
        puzzleHash = _puzzleHash;
    }

    function submitSolution(uint256 guess) public payable {
        require(msg.value == 1 ether, "Submitting solution requires 1 ETH");
        uint256 balance = address(this).balance;
        require(balance != 0, "Game is over. Better luck next time!");

        bytes32 userSubmission = keccak256(abi.encode(guess));
        if (userSubmission == puzzleHash) {
            (bool success, ) = msg.sender.call{value: balance}("");
            require(success, "Transfer failed");
        }
    }
}
```

We can see a simple puzzle game in the contract shown in listing 3.1. The contract has been initialized with a secret number, of which only the hash is visible as the variable `puzzleHash`. By calling the `submitSolution` function, users can submit their `guess` while paying a fee. If the user guesses the correct number (`userSubmission == puzzleHash`), the user receives the contract balance as a reward. Suchff a contract is vulnerable to a displacement attack as an attacker can observe a correct solution being submitted and then submit their transaction with a higher gas price to displace the original transaction.

Insertion

> An attacker A observes a profitable transaction T_V of victim V and submits their own two transactions T_{A_1} and T_{A_2} where T_{A_1} has a higher gas price than T_V and T_{A_2} has a lower gas price so that proposers will include T_{A_1} before T_V and T_{A_2} after T_V [25].

Insertion attacks (often referred to as *sandwich attacks*) involve an attacker making two transactions to sandwich a victim's transaction. The attacker's first transaction has a higher gas price than the victim's, and the second one has a lower gas price. A typical version of this attack is so-called *slippage skimming*. Users can specify a range of allowable price deviation on decentralized exchanges, commonly referred to as *slippage*

range. An attacker can then use this setup to their advantage by purchasing an asset at the lower end of the range and immediately reselling it at the higher price the victim offers. Since slippage is a fundamental aspect of decentralized exchanges, this attack relies more on the exchange mechanisms than the Smart Contract itself.

Listing 3.2: Contract susceptible to insertion by Zhang et al. [25] (Syntax errors fixed for compilability)

```
pragma solidity ^0.8.0;

contract Swap {
    Pair pair;

    function swap(uint amount0) public {
        // Charge constant swap fee
        amount0 = amount0 - 100 gwei;

        pair.token0().transferFrom(msg.sender, address(this), 100 gwei);

        // Calculate amount of token1 swapped to
        uint amount1 = (pair.reserve0() * pair.reserve1()) /
            (pair.reserve0() + amount0) -
            pair.reserve1();

        // Swap tokens
        pair.doSwap(amount0, amount1);
    }
}

contract Pair {
    uint public reserve0;
    uint public reserve1;
    ERC20 public token0;
    ERC20 public token1;

    function doSwap(uint amount0, uint amount1) public {
        // Update reserve of token pair
        reserve0 += amount0;
        reserve1 -= amount1;

        // Transfer tokens
        token0.transferFrom(tx.origin, address(this), amount0);
        token1.transferFrom(address(this), msg.sender, amount1);
    }
}
```

The contract `Swap` shows a simplified version of a DEX used for swapping tokens. The contract `Pair` holds reserves `reserve0` and `reserve1` of the two tokens to be swapped (`token0` and `token1`). After invoking the `swap` function, the swapped amount `amount1` is calculated using the reserves in the `Pair` contract. If an attacker succeeds in invoking the `swap` function before the victim, the reserve values will be modified, resulting in the

victim receiving fewer tokens than expected. The attacker can then sell the tokens at a higher price [25].

Suppression

> An attacker A observes a profitable transaction T_V of victim V and broadcasts numerous transactions to prevent T_V from being included in the same block [25].

Suppression (or *block-stuffing*) attacks involve an attacker making multiple transactions with a higher gas price to prevent a victim's transaction from being mined in the same block. The mechanism exploited here is a so-called *gas limit* imposed on blocks by the protocol, which dictates the maximum amount of gas that can be spent in a single block. Therefore, an attack of this kind requires significant financial resources and is rarely observed on the blockchain.

Torres et al. [27] differentiate between three different approaches to suppression attacks:

1. **Controlled Gas Loop Attack**: The attacker uses a Smart Contract that repeatedly executes instructions in a loop to consume gas, but each loop iteration is checked against the remaining gas limit. Before all gas is consumed, the loop is broken to avoid an out-of-gas exception, rendering the attack much less obtrusive.
2. **Uncontrolled Gas Loop Attack**: An uncontrolled attack, on the other hand, does not check the remaining gas limit and loops indefinitely until all gas is consumed.
3. **Assert Suppression**: This approach is much more subtle and does not need to run infinite loops, as shown in listing 3.3. In Solidity, two ways exist to raise errors during execution: *revert* and *assert*. Up until version 0.8.0 of the Solidity compiler, these two behaved differently in their outcomes. While *revert* returned the unused gas to the caller, *assert* consumed the entire gas limit initially specified. Thus, an attacker was able to consume the entire gas limit with a single instruction. In Solidity 0.8.0 and later, changes have been made so that both instructions now revert and return the unused gas to the caller.

Listing 3.3: Suppression by using assert

```
1  pragma solidity ^0.7.6;
2
3  contract AssertSuppression {
4      function attack() public {
5          assert(false);
6      }
7  }
8
9  contract SuppressionAttacker {
10     AssertSuppression public suppressionContract;
```

```
11
12      constructor(address _targetAddress) {
13          suppressionContract = AssertSuppression(_targetAddress);
14      }
15
16      function attackWithBlockGasLimit() public {
17          uint256 gasLimit = block.gaslimit;
18          // Execute attack with block gas limit
19          suppressionContract.attack{gas: gasLimit}();
20      }
21  }
```

The contract `Suppression` in listing 3.4 shows a simple auction contract vulnerable to suppression attacks. Users can submit their bids by calling the `bid` function and specifying the amount of Ether they are willing to pay. The address of the highest bidder will be stored in the `highestBidder` variable. The auction has a time limit set during the initialization of the contract `block.timestamp + _biddingTime`. A suppression attack could then be mounted a few blocks before the end by an attacker who makes themselves the highest bidder and then fills subsequent blocks with expensive transactions to prevent anyone else from outbidding them [25].

Listing 3.4: Contract susceptible to suppression

```
1  pragma solidity ^0.8.0;
2
3  contract Suppression {
4      address public owner;
5      address public highestBidder;
6
7      uint256 public auctionEndTime;
8      uint256 public highestBid;
9
10     bool public ended;
11
12     constructor(uint256 _biddingTime) {
13         auctionEndTime = block.timestamp + _biddingTime;
14         owner = msg.sender;
15     }
16
17     function bid() public payable {
18         require(block.timestamp <= auctionEndTime, "Auction already ended");
19         require(msg.value > highestBid, "There already is a higher bid");
20
21         if (highestBidder != address(0)) {
22             // Refund the previous highest bidder
23             (bool success, ) = highestBidder.call{value: highestBid}("");
24             require(success, "Transfer failed");
25         }
26
27         highestBidder = msg.sender;
28         highestBid = msg.value;
```

```
29      }
30
31      function endAuction() public {
32          require(block.timestamp >= auctionEndTime, "Auction not yet ended");
33          require(!ended, "Auction already ended");
34          ended = true;
35
36          // Transfer the highest bid to the contract owner
37          (bool success, ) = owner.call{value: highestBid}("");
38          require(success, "Transfer failed");
39      }
40  }
```

3.1.1 ERC-20 Approval

The ERC-20 token standard is by far the most common for Ethereum tokens. To create an ERC-20 token requires the implementation of the ERC-20 interface containing the following functions:

Listing 3.5: ERC-20 interface

```
1   interface IERC20 {
2       function totalSupply() external view returns (uint256);
3
4       function balanceOf(address account) external view returns (uint256);
5
6       function transfer(address to, uint256 value) external returns (bool);
7
8       function allowance(
9           address owner,
10          address spender
11      ) external view returns (uint256);
12
13      function approve(address spender, uint256 value) external returns (bool);
14
15      function transferFrom(
16          address from,
17          address to,
18          uint256 value
19      ) external returns (bool);
20  }
```

The `approve` function allows a user to set an allowance for a spender to use the tokens on their behalf. This contract is susceptible to so-called *approval front-running* attacks which basically follows the following steps [23]:

1. A user allows the address `spender` to transfer x tokens on their behalf by calling `approve(spender, x)`.
2. At a later point in time, they adjust the allowance to y by calling `approve(spender, y)`.

3. Before the second transaction is included in a block, an attacker can initiate the `transferFrom` function to transfer x tokens from the user's account.

4. If this transaction is successfully processed, the attacker can then use the second `approve` transaction to transfer y more tokens from the user's account, resulting in $x + y$ tokens being transferred instead of $max(x, y)$ tokens.

This attack is possible since the `approve` method overwrites the current allowance regardless of whether it has already been used.

Listing 3.6: Implementation of the approve function in the OpenZeppelin ERC20 implementation [28]

```
function _approve(
    address owner,
    address spender,
    uint256 value,
    bool emitEvent
) internal virtual {
    if (owner == address(0)) {
        revert ERC20InvalidApprover(address(0));
    }
    if (spender == address(0)) {
        revert ERC20InvalidSpender(address(0));
    }
    _allowances[owner][spender] = value;
    if (emitEvent) {
        emit Approval(owner, spender, value);
    }
}
```

On the other hand, the `_spendAllowance` function merely verifies the *current* allowance while not tracking transferred funds before the last `approve` call [29].

Listing 3.7: Implementation of the spendAllowance function in the OpenZeppelin ERC20 implementation [28]

```
function _spendAllowance(
    address owner,
    address spender,
    uint256 value
) internal virtual {
    uint256 currentAllowance = allowance(owner, spender);
    if (currentAllowance != type(uint256).max) {
        if (currentAllowance < value) {
            revert ERC20InsufficientAllowance(spender, currentAllowance,
    value);
        }
        unchecked {
            _approve(owner, spender, currentAllowance - value, false);
        }
    }
}
```

3.1.2 Detection

The detection of front-running attacks is notoriously tricky as there exists no strict way to distinguish between an attacker and a user submitting a benign transaction to a specific contract. Therefore, detecting front-running on a transaction level relies heavily on heuristics, e.g. proposed by Torres et al. [27] in the following ways:

Displacement

Having two transactions T_V and T_A, for **displacement** attacks:

1. Sender and recipient are different for both T_A and T_V.
2. The gas price of T_A is higher than the gas price of T_V.
3. The transactions must be identical (or display a degree of similarity above a certain threshold).

Insertion

Insertion (or sandwich) attacks require a more sophisticated approach. Furthermore, the authors have limited detection to DEXs. The basis for the heuristics is the *event* E, defined as a tuple of the following form:

$$E = (s, r, a, c, h, i, g)$$

where s is the sender, r is the recipient, a is the amount, c is the contract address, h is the hash of the transaction, i is the index of the transaction in the block, and g is the gas price.

Let there be three events E_{A1}, E_{A2}, E_V. Detection relies on the following heuristics:

1. Tokens are transferred from the same exchange to A in E_{A1} and to V in E_V. The tokens transferred in E_{A2} are the tokens received by A in E_{A1}.
2. The number of tokens bought in E_{A1} roughly corresponds to the number of tokens sold in E_{A2}.
3. The token contract addresses are identical for all events: $c_{A1} = c_{A2} = c_V$.
4. The transaction hashes for each event are different: $h_{A1} \neq h_{A2} \neq h_V$.
5. The transaction indices must be strictly increasing: $i_{A1} < i_V < i_{A2}$.
6. The gas prices must be strictly decreasing: $g_{A1} > g_V > g_{A2}$.

Suppression

A much simpler heuristic can be applied for detecting **suppression** attacks:

1. Cluster transactions in one block with the same receiver.
2. All transactions in any given cluster must have consumed more than 21,000 gas units as lower gas consumption is found in transactions that only transfer value and do not execute any code.
3. The ratio between gas used and gas limit must be greater than 99%.

CHAPTER 4

Technical Definition

A vulnerability such as front-running is not easily defined as it depends on multiple factors, such as the application logic, the state of the blockchain, the gas price, the flow of information, and the behavior of proposers. These aspects make a concise and formal definition difficult. Several attempts exist to define and categorize front-running vulnerabilities, which we discuss in the remainder of this section (see section 4.3).

4.1 Community Approaches

One of the most comprehensive and rigorous attempts to classify and define Smart Contract vulnerabilities is the Smart Contract Weakness Classification (SWC) registry [30]. This registry was an open-source effort to collect, define, and adequately categorize possible vulnerabilities in Ethereum Smart Contracts. At the time of writing, it contains 36 vulnerabilities, each linked to a Common Weakness Enumeration (CWE).

The SWC registry represents a significant contribution to Smart Contract security research. However, it is essential to note that the project has been abandoned and has not received any updates since 2020 [31]. Since then, new developments in Smart Contract security and vulnerability research have been made, but these still need to be reflected in the SWC registry.

TOD has already been defined in the registry under the ID *SWC-114* [23]. The vulnerability is described as a race condition, a common issue in concurrent programming where the outcome of a process depends on the sequence of events outside the creator's control.

4.2 Systematic Literature Review

By systematically reviewing the literature, we have identified several definitions of front-running vulnerabilities with varying degrees of formality. To do this, we identified four

primary domains for our search:

- Ethereum
- Front-running
- Automated Detection
- Vulnerability

For each domain, we developed a comprehensive list of keywords and their variations to ensure broad coverage and subsequently combined them into queries for each database.

```
    ("Ethereum" OR "smart contract" OR "smart contracts")
AND (
    "front-running" OR "front-running" OR "front running"
    OR "TOD" OR "Transaction Order Dependence" OR "Transaction Order
    Dependency"
    OR "Race condition")
AND (
    "tool" OR "tools"
    OR "automated" OR "automatic" OR "automatically"
    OR "detect" OR "detecting" OR "detected" OR "detection" OR "detector"
    OR "analysis" OR "analysing" OR "analyzed" OR "analysed" OR "analyser" OR
     "analyzer"
    OR "identify" OR "identifies" OR "identifying")
AND (
    "vulnerability" OR "vulnerabilities"
    OR "weakness" OR "weaknesses"
    OR "bug" OR "bugs"
    OR "security risk" OR "security risks"
    OR "security issue" OR "security issues"
    OR "insecurity" OR "insecurities")
```

The following databases were used:

- ACM Digital Library
- IEEE Xplore
- ScienceDirect
- Scopus
- Google Scholar

4.2.1 Initial Screening

Our initial searches yielded around 2000 potentially relevant publications. We then manually screened the results for the following criteria:

- Focus on Ethereum smart contract vulnerabilities
- Discussion of automated detection methods

- Published in English
- Peer-reviewed academic publications or high-quality technical reports

While doing so, quality criteria were applied to ensure the quality of the results and refine our selection, including:

- Clarity of research objectives
- Soundness of methodology
- Validity of results and conclusions
- Relevance to our research questions

4.2.2 Final Selection for Definitions

In the final iteration, we selected 86 papers dealing with the detection, mitigation, prevention, or analysis of front-running vulnerabilities in one form or another. These papers were used to extract the following definitions of front-running vulnerabilities.

4.2.3 Limitations

While our review aimed to be comprehensive, we acknowledge potential limitations:

- Possible exclusion of relevant non-English publications
- Recent developments may not have been captured
- Potential bias in database selection and search term formulation

4.3 Definitions

The following section will provide an overview of the definitions encountered during the literature review. Few papers provide a rigorous definition, while most others provide an informal description of the issue or the detection logic. The vast majority of papers do not provide a definition at all.

4.3.1 Notation

During the review, several notations were encountered that defined front-running vulnerabilities. We will use one common notation to present the different definitions for better comparison. Our notation is heavily inspired by Ma et al. [32] and will be used throughout the following sections.

1. A contract instance, denoted as $\mathcal{C}$, is defined as a five-tuple

$$\mathcal{C} = \langle address, balance, \mathcal{V}, \mathcal{F}, \sigma_0 \rangle$$

where *address* is the contract address, *balance* the balance of the contract, $\mathcal{V}$ the set of storage variables, $\mathcal{F}$ the set of functions and σ_0 the initial (world) state.

2. A transaction T is defined as a tuple $T = \langle a, b, f(\vec{x}) \rangle$ where a is the transaction submitter, b the transaction receiver, f the function invoked by the transaction and $\vec{x}$ the arguments passed to the function.

3. We furthermore denote the sets of storage variables that are read or written to by a transaction T as R_T and W_T, respectively, with $R_T, W_T \subseteq \mathcal{V}$.

4. States will be denoted as σ, while the states following a specific transaction sequence T_i, T_j will be denoted as σ_{T_i,T_j}.

5. A state transition of a transaction T on state σ will be denoted as $\sigma \xrightarrow{T} \sigma'$.

6. A *sequence* $\mathcal{H} = [T_1, T_2, ..., T_n]$ is a sequence of n transactions. A state transition of a sequence $\mathcal{H}$ on state σ will be denoted as $\sigma \xrightarrow{\mathcal{H}} \sigma'$. The set of all possible sequences is denoted by $\mathbb{H}$. A transaction at position k of $\mathcal{H}$ is denoted as $\mathcal{H}[k]$.

7. Two sequences, $\mathcal{H}$ and $\mathcal{H}'$, are equivalent, written as $\mathcal{H} \equiv \mathcal{H}'$, if they contain the same set of function invocations.

8. A sequence can be permuted by a transformation function $\mu : \mathbb{H} \to \mathbb{H}$ that accepts a sequence $\mathcal{H}$ and transforms it to an equivalent sequence $\mu(\mathcal{H})$.

9. A participant who submits a transaction is denoted as a.

10. The assets (any form of value, e.g., Ether, tokens, NFTs) which a transaction submitter, a, possesses are denoted as $\mathcal{A}(a)$. The value of the assets owned by a at state σ is denoted as $\mathcal{A}_\sigma(a)$.

4.3.2 Formal definitions

In the following sections, we will provide an overview of formal definitions of front-running vulnerabilities as proposed in the literature.

Bose et al. (Sailfish) [33]

The authors of Sailfish, a tool for detecting so-called *state-inconsistency* bugs, define TOD by introducing the notion of state-inconsistency [33]. A state-inconsistency bug for a contract instance $\mathcal{C}$ with the initial state σ_0 is defined as follows.

Let *sequence* $\mathcal{H} = [T_1, T_2, ..., T_n]$ [1] be a sequence that can be executed by the EVM, and let σ' be the final contract state reached by executing the transactions in order on the initial states, $\sigma_0 \xrightarrow{\mathcal{H}} \sigma'$.

[1] The authors in their paper refer to these transactions as *events*.

If there exists a sequence $\mathcal{H}_2 = \mu(\mathcal{H}_1)$, such that $\sigma \xrightarrow{\mathcal{H}_1} \sigma_1$ and $\sigma \xrightarrow{\mathcal{H}_2} \sigma_2$ for $\sigma_1 \neq \sigma_2$, then $\mathcal{C}$ is said to have a state-inconsistency bug.

In this definition, the authors do not explicitly mention the term TOD but instead refer to the more general notion of state inconsistency. Furthermore, no reference is made to financial damage, which others consider a crucial characteristic of front-running vulnerabilities.

A closely related (but less formal) definition can be found in the work of Kolluri et al. [34] considering the notion of *event-ordering bugs* that can also be seen as a generalization of TOD vulnerabilities. In their definition, the authors refer to sequences as *traces* and define an event-ordering bug as a pair of valid *traces*, which are permutations of each other but lead to different final states.

Ma et al. [32]

In [32], the authors examine *transaction races* in Ethereum Smart Contracts. They do not formally define TOD but rather describe it as a form of transaction race relevant only to "[...] balance differences, whereas transaction race bugs are more general because both balance differences and contract storage differences are considered" [32].

Still, they provide a formal definition of transaction races, which applies to TOD vulnerabilities. For a transaction race to occur, there must be a contract state σ_i at which a sequence $\mathcal{H} = [T_1, T_2, ..., T_n]$ exists, and the following conditions hold:

1. $\sigma_0 \xrightarrow{T_1} \sigma_1 \xrightarrow{T_2} \sigma_2$ and $\sigma_0 \xrightarrow{T_2} \sigma_1' \xrightarrow{T_1} \sigma_2'$.
2. $(R_{T_1} \cap W_{T_2}) \cup (W_{T_1} \cap R_{T_2}) \cup (W_{T_1} \cap W_{T_2}) \neq \emptyset$
3. $\sigma_2 \neq \sigma_2'$

R_{T_i} and W_{T_i} are the sets of storage variables that are read or written to by T_i respectively.

In this case, again, the definition does not consider financial damage. However, the authors do refer to the critical relationship between transactions and storage variables, which are an essential indicator for many tools to detect TOD vulnerabilities such as Mythril [35] or Securify [36].

Zhang et al. (Nyx) [37]

The authors of this paper have developed a tool called Nyx for specifically detecting TOD vulnerabilities. At the time of writing, the tool is not publicly available. The authors define TOD vulnerabilities as follows:

Let f_1, f_2 be a pair of functions where $f_1, f_2 \in \mathcal{F}$. This pair of functions is vulnerable to TOD if and only if there exist two transactions T_1, T_2, a state σ and two distinct transaction submitters $a_1, a_2 \wedge (a_1 \neq a_2)$ where:

$$T_1 = \langle a_1, f_1(\vec{x}_1)\rangle, \quad T_2 = \langle a_2, f_2(\vec{x}_2)\rangle$$
$$\sigma \xrightarrow{T_1,T_2} \sigma_{T_1,T_2}, \quad \sigma \xrightarrow{T_2,T_1} \sigma_{T_2,T_1}$$

and the following condition holds:

$$\mathcal{A}_{\sigma_{T_1,T_2}(a_1)} > \mathcal{A}_{\sigma_{T_2,T_1}(a_1)} \wedge \mathcal{A}_{\sigma_{T_1,T_2}(a_2)} < \mathcal{A}_{\sigma_{T_2,T_1}(a_2)}$$

The authors do not use the notion of sequences, but the definition can also be applied to sequences. Furthermore, they do not refer to storage variables.

This definition emphasizes that a front-running vulnerability needs to result in financial damage to a participant as expressed by the condition that the assets of a_1 have increased while the assets of a_2 have decreased. A similar but less rigorous definition that does not require financial damage has been proposed by Luu et al. [38] and Wang et al. [39].

Tsankov et al. (Securify) [36]

Securify, a tool for detecting security vulnerabilities in Smart Contracts, uses a pattern-based approach to detect various vulnerability types. It does so by decompiling the EVM bytecode into a stackless representation and inferring semantic facts regarding data and control flows. Facts may be, e.g., `MayDepOn(b, dataload)`, which expresses that the value of variable `b` may depend on the value returned by `dataload` [36].

These facts are then used to check against predefined patterns using a domain-specific language. For TOD vulnerabilities, the tool checks for the following pattern as a violation.

If a function call is made that transfers some amount of Ether:

```
some call(_, _, _, Amount).
```

A storage variable X_1 is loaded into the variable Y:

```
some sload(_, Y , X1).
```

Some value is written to the storage location X_2:

```
some sstore(_, X2, _).
```

Finally, a check is made if the amount transferred is determined by Y and the storage location being read from (X_1) and written to (X_2) is the same, and the value of the storage location X_1 is constant:

$$\texttt{DetBy}(\texttt{Amount}, Y) \wedge X_1 = X_2 \wedge \texttt{isConst}(X_1).$$

The tool Mythril uses a related approach for detecting TOD vulnerabilities. It does so by checking if the called value is dependent on storage variables or the contract balance [35].

4.3.3 Non-formal definitions

This section will provide an overview of non-formal definitions of front-running vulnerabilities as proposed in the literature. They are generally less rigorous and provide a more general overview of the vulnerability. In some cases, no definition is provided but rather a description of the detection logic.

Zhang et al. [25]

The authors define front-running to consist of a tuple of three transactions $\langle T_a, T_v, T_a^p \rangle$ where T_v is the victim's transaction and T_a, T_a^p are the attacker's transactions. For an attack scenario, the sequence of transactions $\mathcal{H}_a$ must be such that T_a is included in the block before T_v and T_a^p is included after T_v. So $\mathcal{H}_a = [T_a, T_v, T_a^p]$ while a benign sequence would be $\mathcal{H}_b = [T_v, T_a, T_a^p]$.

This sequence of transactions is considered an attack only if it satisfies two properties:

1. **Attacker gain:** The attacker benefits financially when the sequence $\mathcal{H}_a$ is executed compared to $\mathcal{H}_b$.

2. **Victim loss:** The victim suffers a financial loss when the sequence $\mathcal{H}_a$ is executed compared to $\mathcal{H}_b$.

Li [40]

In their paper, the author does not refer to TOD or FR but is examining the impact of a more generic concept they have named *concurrency exploits*. The definition provided matches the definition by Ma et al. to a large extent section 4.3.2 with two transactions reading and writing from the same storage variable.

A noteworthy difference is that in this definition, only so-called *security-critical* operations that depend on the accessed storage variable are considered. These are four specific EVM opcodes: `CALL`,`CALLCODE`, `DELEGATECALL`, `SELFDESTRUCT`

M. Fu et al. [41]

The authors considered the following logic for detecting TOD vulnerabilities in the context of critical paths:

For all external calls, find all storage variables that can affect the call value or call address; they call these variables *interesting storage variables* or **Instor**. For each of these, determine if they are likely to change. If a `SSTORE` operation can modify the variable, a TOD vulnerability exists. Other researchers have applied similar detection logic [42].

ZEUS [43]

The authors of this paper briefly mention transaction order dependence in their taxonomic analysis of vulnerabilities. It is worth pointing out that they do not consider it a vulnerability but merely a *limitation* of the Ethereum blockchain.

Detection is performed by "[...] determining potential read-write hazards for global variables that can influence Ether flows". Determining these flows is done by *tainting* all global variables written to and then checking if this taint is propagated to a `send` or `call` [43].

VulHunter [44]

As opposed to ZEUS, the authors of VulHunter consider TOD a vulnerability with high severity.

The paper classifies TOD vulnerabilities into three categories: key storage variable, owner authentication, and approved tokens. These vulnerabilities occur when the state variables are assigned and used in different functions, and the order of transactions invoking these functions can be altered within the same block, leading to incorrect contract behavior [44].

Munir et al. [45]

The authors of the *TODLER* tool for detecting TOD vulnerabilities have subdivided the vulnerability into four distinct categories:

1. **TOD-Amount (TA)** - This type occurs when the reordering of transactions causes a different amount to be transferred. Usually, this occurs if the amount is defined in a state variable. Therefore, contracts having public functions that allow changes to such values are vulnerable to this type.
2. **TOD-Ownership (TO)** - Analogous to the previous type, with the difference that instead of the amount transferred being altered, the transfer recipient is changed.
3. **TOD-Transfer (TT)** - Usually transactions are validated before being executed. If these validations contain variables that can be updated by public functions, the transfer is indeterministic.
4. **TOD-Selfdestruct (TS)** - This vulnerability can occur with the `selfdestruct` function. This type of vulnerability can be considered obsolete because this instruction has been modified substantially in a recent update [46] of the Ethereum protocol.

Ramakrishnan et al. [47]

The authors of this paper merely refer to TOD as a race condition that occurs mainly in ERC20 tokens. They do not provide a formal definition of the vulnerability.

Wang et al. [48]

In their work, Wang et al. provide a detailed examination of so-called *nondeterministic payment bugs*, which are bugs resulting from the inherent nondeterminism of the blockchain. Furthermore, they consider these bugs the root cause for many common vulnerabilities, including TOD.

Qin et al. [49]

The authors refine the notion of front-running by adding two distinctions:

1. **Destructive Front-Running** – If the attacker's transaction is included before the victim's and causes the victim's transaction to fail. Attacks of this type do not consider the impact on subsequent transactions.
2. **Tolerating Front-Running** – If the attacker's transaction is included before the victim's and needs the victim's transaction to succeed (e.g., for sandwich attacks). For attacks of this type, profit is impossible if the victim's transaction fails.

4.4 Discussion

As noted before, the definitions vary considerably between papers. An indicator of this discrepancy can be observed for fundamental considerations such as if FR is even considered a vulnerability or merely a limitation as stated by [43].

The formal definitions can be considered extensions of each other. While the authors of Sailfish [33] provide a general definition of so-called *state-inconsistency bugs*, which can best be described as *two sequences of transactions leading to a different final state*, Ma et al. [32] extend this definition by considerations regarding storage variables which are being read and written. While Zhang et al. [37] do not consider the storage variables in their definitions, they are the only ones that propose taking into account the assets of the participants. Therefore, in this case, a vulnerability only manifests if it results in actual financial damage.

Regarding non-formal definitions, the authors provided descriptions with varying degrees of detail and rigor. In some cases, the definition can be considered too broad or misleading, as is the case in the paper by Yuan et al. [50], which describe FR as *"Two dependent transactions that invoke the same contract are included in one block."*

In many cases, the definitions are rather descriptions of the underlying system, which allows for its occurrence. Descriptions of this kind contain one or more of the following aspects:

1. Once a transaction is submitted to the blockchain, one can not exactly predict when it will be executed.

2. The time of execution depends on the proposer selecting the transactions to include in the next block.
3. Attackers (mostly proposers) can use this unpredictable behavior to their advantage.
4. Proposers can use their advantageous position to collude.

Descriptions that match this pattern appear in several examined works [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67].

In one case, the definition was further restricted to specific cases only. In the paper by Cai et al. [68], the weakness is referred to as *ERC20-TOD* and explained only in the context of ERC20 tokens.

More direct descriptions of the vulnerability occur but frequently restrict themselves to only a few sentences, such as the description by Dai et al. [69], which offers one sentence in their taxonomic enumeration of vulnerabilities.

> *"The execution order of multiple transactions generated by the same smart contract will cause differences in the final state of the smart contract."*

This description can be considered more comprehensive than the one offered by Yang et al. [70] (*"The result of contract execution is related to the transaction order"*) or Liao et al. [71] (*"State will depend on the tx order"*).

Wang et al. [72] have subsumed their definition of TOD under a common vulnerability "*TD&TOD*", which also includes *timestamp dependence*. We would argue that these vulnerabilities are different enough to merit separate categories and definitions.

However, we encountered many papers researching TOD, in most cases discussing its detection, and not providing any definition whatsoever: [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [83] [84] [85] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [99] [100] [101] [102] [103] [104] [105] [106] [107] [108] [109] [110] [111] [112] [113] [114].

CHAPTER 5

Maximum Extractable Value and Front-Running

An important concept for understanding FR is the idea of MEV[1] introduced by Daian et al. [16] MEV is a metric which defines how much value one can extract from any given contract through all possible re-orderings of the transactions. Thus, a miner (or validator) interested in maximizing their profits will try to arbitrarily include, exclude, or reorder transactions within the blocks they produce [52]. It is important to note that optimizing MEV is common practice in the blockchain ecosystem and is generally considered neither unethical nor malicious in its own right. Differentiating between harmful and benign actions regarding activities involving MEV is not straightforward. It is usually done by considering whether or not the attack results in price degradation for the user [16]. This differentiation is sometimes called *toxic* and *non-toxic* MEV [116].

So-called MEV-bots are automated agents constantly scouring the blockchain for profitable opportunities. These agents are referred to as *searchers* [117]. By observing the *mempool*, they can quickly react to new transactions and maximize their profits through different strategies. These bots led to the conceptualization of Ethereum as a so-called *dark forest* [118]. It describes a situation in which mere detection is equal to financial damage. The authors who coined this term refer to an incident where they could not recover lost funds for a user directly as they would have been front-run by a bot and, therefore, had to resort to sophisticated obfuscation methods to achieve their goal [118]. This term, borrowed from a science-fiction novel, emphasizes the adversarial conditions under which participants have to operate.

Many attacks on the Ethereum blockchain are related to MEV bots. According to some authors, these bots pose a substantial risk to the blockchain system as they could be a destabilizing force. Estimates suggest that between 2020 and 2023, the total MEV has

[1] In some cases referred to as *maximum extractable value* [52] or *blockchain extractable value* [115].

amounted to an estimated $550-$650 million on Ethereum alone [116]. It is therefore worthwhile to provide insights into the impact of MEV and front-running.

5.1 MEV strategies

MEV-bots employ various attacks apart from FR for extracting value from the blockchain. In the following section the most common strategies are discussed. The definitions are based on [119].

5.1.1 Sandwich Attacks

For this attack, the attacker places two transactions immediately before and after the victim's transaction (see section 3.1). A typical example would be to buy an asset before the victim and sell it immediately after the victim's transaction has raised the price.

Formally, a sandwich attack can be defined as follows.

MEV Sandwich. *Let there be three transactions: a victim transaction T_V and two sandwich transactions T_1 and T_2, originating from the same address. We furthermore need to define that the transactions T_V and T_1 are exchanges from currency C_1 to currency C_2, while T_2 is an exchange from C_2 to C_1. A sandwich MEV occurs if $T_1 \prec T_V \prec T_2$.*

Such attacks are common, especially in AMM protocols, as the attacker is aware of the price changes that will occur due to a significant transaction.

Attacks of this type can be reasonably complex and lucrative, as seen in the transaction `0x1bf62ad5647ddf7a289cb22edba7a2fea707d45def2f5a3a335c0e5e4df446bd` [2] resulting in more than $40.000 in profits for the attacker and involving 18 different tokens.

However, these attacks also bear considerable risks for attackers. Given that the attacker pays significant gas fees for the transactions, the attack can easily result in a net loss if prices do not change as expected or the order of transactions does not go as planned [120].

5.1.2 Backrunning

Backrunning is a strategy that involves placing a transaction directly *after* the victim's transaction. Suppose a transaction is expected to significantly impact the asset price (e.g., a large buy order). In that case, the attacker can buy the asset from another exchange and sell it back to the current exchange, thus profiting from the price difference [117].

[2] For the entire sequence see `https://www.eigenphi.io/mev/eigentx/0x075ed8820cabd7da44aa603a9bfc7a669bcb82b98ace3105671a392a7791340d,0xf83d22f8759917e1bc5a1c320079b64a251b9a243c341beb4e6cff3f93504717,0x1bf62ad5647ddf7a289cb22edba7a2fea707d45def2f5a3a335c0e5e4df446bd?mevId=0x91a76229be2ef268984954f71d0b9a705c667ea87f90766fdc36ba9be49e1b0e`

5.1.3 Arbitrage

Arbitrage is generally considered the most benign use case for MEV bots. However, depending on the specific strategy employed, one could consider it malicious. Classical (or passive) arbitrage involves monitoring the price of an asset on multiple markets and executing a trade if the expected profit surpasses the expected costs. However, arbitrage can also be performed more proactively by scanning and front-running pending transactions for existing arbitrage transactions. Formally, proactive arbitrage is described as follows:

MEV Arbitrage. *Let V be an unpublished victim arbitrage transaction in the public mempool, exchanging currency C between exchanges E_1 and E_2, denoted as $E_1(C) \rightarrow E_2(C)$. Let $P(E_n, C)$ be the price of C on exchange E_n. Arbitrage occurs if $P(E_1, C) + F_{vic} < P(E_2, C)$, where F_{vic} is the victim's mining fee. An MEV extractor can submit the same transaction with a higher fee $F_{mev} > F_{vic}$, as long as $P(E_1, C) + F_{mev} < P(E_2, C)$ still holds.*

A more complex strategy involves scanning the mempool for transactions that are large enough so that one can expect that the asset price will be affected by it. In anticipation of this, the attacker can buy the asset from another exchange and sell it back to the first one, thereby profiting from the difference. Hence, the difference between sandwich attacks and malicious arbitrage in the execution pattern is small.

5.1.4 Liquidation

An essential aspect of DeFi is the possibility of decentralized lending and borrowing, widely considered one of the most successful applications in decentralized finance [121]. Lenders aim to profit by participating in *lending pools* where borrowers can borrow assets by agreeing to pay interest and offering collateral. An important mechanism in this regard is that the value of the collateral can fluctuate dramatically. If the price of the given collateral drops below a threshold (and is therefore worth less than the borrowed asset), the loan can be liquidated.

This liquidation can be performed by *anyone*, meaning that liquidation can be described as a situation in which a user *who is not the borrower* repays the debt on behalf of the borrower and, in turn, receives the collateral at a discount. Loans available for liquidation can be recognized easily as lending pools mark such loans as "unhealthy" [119].

Formally, the liquidation process can be described by the following process:

MEV Liquidation. *A collateral C is leveraged against a loan with the value v_L. The current price of the collateral at time t is $P(C, t)$. The value of the collateral when the loan is taken out is $P(C, t_0) = v_L$. The loan is considered unhealthy if $P(C, t_n) < v_L$ for some $t_n > t_0$.*

It is apparent why liquidations are generally seen as beneficial for market efficiency. Lenders avoid financial losses due to price volatility, and liquidators can profit by selling the collateral on the market.

Liquidations come in two different types:

- **Fixed spread-based** – Liquidations are settled within one transaction following the First-Come-First-Serve (FCFS) principle.
- **Auction-based** – Liquidations are settled by auction, where the highest bidder receives the collateral.

The first type constitutes an attractive target for MEV bots. Strategies are manifold but generally involve scanning the current mempool for either pending liquidations, which can be front-run, or spotting transactions that could potentially create liquidation opportunities in the future [119].

5.2 Effects of MEV Extraction

MEV bots affect the blockchain in various ways. While it cannot be said that all MEV extraction is harmful, several negative implications are worth mentioning.

Financial Damage

The primary impact on the network is the financial damage caused by (malicious) MEV extraction within the DeFi ecosystem. Although it is a fiercely debated ethical question whether any given MEV extraction is harmful, generally speaking MEV extraction is capturing value from often unsuspecting users. Research conducted by EigenPhi [122] suggests that between July and August 2024 alone, the value extracted only through sandwich attacks has surpassed $10 million.

Network inefficiency

The prevalence of MEV extraction has in the past led to increased network congestion and transaction fees. If multiple front-runners are competing for the same transaction, this leads to a Priority Gas Auction (PGA) [16] as the bot with the highest transaction fee usually wins. This quickly leads to network congestion as many transactions are competing for limited block space, resulting in higher transaction fees for all participants [123].

Destabilizing consensus

High MEV rewards that surpass miner rewards can potentially undermine consensus mechanisms and lead to validators deviating from honest practices. If MEV rewards grow more profitable than traditional mining rewards, this can lead to several undesirable outcomes: forks in the chain with high-fee blocks to attract other miners, disrupting

chain progression [117]. Another aspect of this is the so-called *time-bandit attack* in which miners rewrite history to steal funds allocated by smart contracts in the past [16].

Centralization

Maximizing MEV requires sophisticated technical capabilities, making it much more likely that institutional operators crowd out individual operators entirely [124]. MEV extraction can thus result in significant economies of scale and cause centralization of a few sophisticated actors extracting most of the value. This is exacerbated by the secrecy surrounding the underlying algorithms in bots. The details of the operational mechanics of MEV bots are difficult to obtain, as the algorithms used in these bots are often proprietary and kept secret. Few outliers exist, such as the *Fast Lane Protocol* [125], developed by Bancor, which is a fully open-source arbitrage protocol with the stated goal of allowing any user to perform arbitrage. Another concern mentioned is that widespread MEV extraction could cause "vertical integration" of miners and traders, which could badly influence the decentralization and transparency of the network by creating closed systems [117].

On the other hand, some MEV extraction is considered beneficial for the network. For example, arbitrage bots can help to stabilize prices across different exchanges and provide liquidity [116].

5.2.1 Quantifying the Financial Impact

Front-running attacks and the concept of MEV extraction have significantly affected the Ethereum network. It is, therefore, essential to quantify these attacks' impact on the network. Some research has been conducted which tried to quantify the financial damage [49] during a period oaf December 2018 and August 2021. Expressed in USD, the researches found that around $540 million were extracted from the network.

`mev-boost` (see section 5.3.2) has been a significant factor in the Ethereum ecosystem. Given that it operates under the stated goal of maximizing MEV extraction and democratizing its rewards, it is essential to quantify the financial dimensions in which it has affected the network.

A representative sample of blocks is selected as a basis for this analysis. We chose approximately 1000000 blocks from block height 19475310 to 20475310 (see chapter A in the appendix for more details). We can calculate diverse metrics using the API provided by `zeromev.org`, a website tracking MEV extraction in each block as they occur. For each block, one can find the transactions that were considered to be MEV transactions and their category (e.g. sandwich attack, arbitrage, etc.).

It is important to note that defining what constitutes a successful MEV extraction is somewhat arbitrary and might vary between different projects. Still, it offers valuable insights into the financial aspects of MEV extraction.

According to the data, around half of the blocks contain successful sandwich attacks, leading to the cumulative user loss in USD for the selected blocks amounting to almost $60 million, as seen in fig. 5.1.

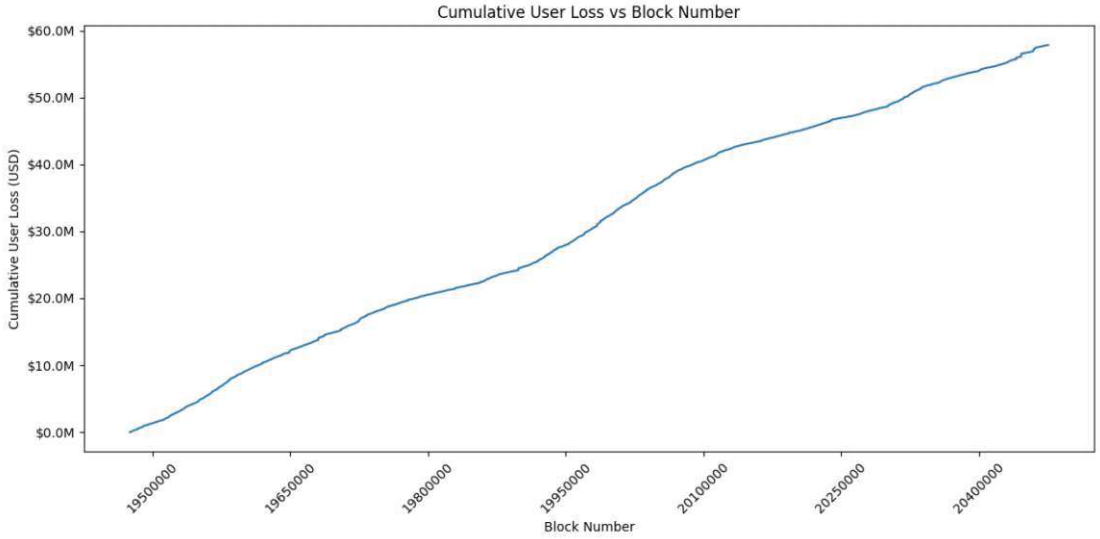

Figure 5.1: Cumulative User Loss vs Block Number

It is worthwhile to inspect the distribution of the user losses.

Loss Category	Attacks	% of Attacks	Loss (USD)	% of Total Loss
$0-$50	356,999	66.38%	$6,638,273.64	11.48%
$51-$100	77,674	14.44%	$5,575,275.34	9.64%
$101-$500	86,131	16.01%	$17,838,974.48	30.85%
$501-$1000	10,163	1.89%	$7,002,591.42	12.11%
$1001-$5000	6,228	1.16%	$11,635,363.62	20.12%
$5001-$10000	384	0.07%	$2,604,382.08	4.5%
$10000+	260	0.05%	$6,535,214.35	11.3%
Attacks	537,839	Total Loss (USD)	$57,830,074.93	

Table 5.1: Distribution of sandwich attacks and associated losses

One can observe that the vast majority of the attacks are in the $0-$50 category. This category accounts for 66.38% of the attacks and 11.48% of the total loss, suggesting that the majority of the attacks performed are small in scale and do not result in significant losses for the users. However, it is still responsible for approximately as much loss as the largest category, with the difference being that the captured value here is divided among merely 0.05% of all the attacks.

Cross-referencing this data with another data source, *libMEV* [126], we can observe the actors involved in the attacks and the degree of centralization that is occurring in this space. To do so, we extracted data for the same period as in the previous analysis (see section A.1.2), collecting data for 50 top attackers, 39 of which turned a profit in the same period (for the data, see table B.2). A few actors are responsible for the majority of the attacks as well as the extracted value. The first six searcher contracts have captured almost 95% of the total profit, suggesting that in terms of democratization of the MEV extraction, it still has a long way to go. Furthermore, one can see that some attackers are very selective regarding their choices, submitting only a few bundles (with some in the list having submitted as few as one bundle). On the other hand, the attacker responsible for by far the most submitted bundles (rank 3 in the list) surpasses other searchers by several orders of magnitude. This searcher contract [127] is presumably the most famous MEV searcher in the space and is estimated to have extracted $13 million [128] since the merge [129].

5.2.2 Effects on the Ethereum Network

Centralization of network actors

Given that the primary answer to combatting the adverse effects of MEV extraction is the implementation and usage of Proposer/Builder Separation (PBS) as implemented in `mev-boost`, it leads to new dynamics in the network since switching to PoS. The share of PBS blocks constantly remains around 90% of the total blocks produced, meaning that one can safely assume that it has become critical infrastructure for the network. [130] The short analysis of the network in the following sections shows that `mev-boost` has been a centralizing force (as argued by Yang et al. [117]) in the ecosystem.

We will focus on three main actors: validators, builders, and relays. Centralization has different implications for each of these actors.

Validators

As mentioned in Ethereum's roadmap towards the implementation of PBS on the consensus layer, the centralization of builders is not considered harmful as long as there is a "[...] robust and maximally decentralized network of validators able to prove the blocks are honest." [124]

However, an analysis by Grandjean et al. [131] concluded that the staking power in Ethereum is as centralized as the mining power during the PoW era, with one staking pool (*Lido* [132]) controlling almost a third of all validators. A possible attack scenario for entities that control validators can be gleaned from the case study of the Peraire-Bueno brothers (see section 5.3.4 for details).

A threat that results from high validator centralization is the so-called *multi-block MEV extraction* [131], which describes a scenario in which value extraction through transaction order manipulation is extended over multiple blocks. Such an attack is possible only

if the validator is certain to be in control of multiple blocks. Sequences of this kind happen regularly and daily, with *Lido* having been observed to control sequences of up to thirteen blocks. Given that the largest share of the profits is captured by validators and the possibility of interrupting the network when controlling more than 33% of the staking power, these findings are concerning [131]. Such attacks are difficult to detect, with the techniques presented in section 3.1.2 being inadequate for detecting them.

Relays

Relays are sometimes considered a single point of failure in the PBS setup, as they are responsible for the distribution of the blocks to the validators. Therefore, a certain degree of trust is required. A much-discussed aspect of relays is their ability to censor transactions [117].

Since the merge, 92% of the blocks distributed by relays have used `mev-boost` [133]. Furthermore, Grandjean et al. have discovered that in all relays they observed, the trust assumptions have not been fulfilled. [131] With the proposed implementation of PBS on the consensus layer, the role of relays is planned to be eliminated. [134]

Builders

The block builder landscape is the most centralized part of the network, dominated by a few large players. Considering all blocks since the merge, six entities have produced approximately 90% of all blocks. However, newer developments led to even higher levels of concentration, with only three entities producing around 95% of all blocks in the time between July and August 2024 [135]. In general, this centralization aspect is not considered harmful to the network. However, aspects regarding censorship resistance need to be taken into account as they can lead to unwanted MEV extraction [131].

5.3 Mitigations

In their paper, Alipanahloo et al. [123] present a comprehensive survey on MEV mitigation on Ethereum and Layer-2 chains. They further present a taxonomy of MEV mitigation approaches, outlining the main ideas and strategies.

As shown in Figure fig. 5.2, there are two primary strategies of MEV mitigation: Prevention/Reduction and Side-effects Reduction. The first strategy focuses on removing or limiting the ability of a validator to reorder transactions, which aims to prevent them from extracting MEV altogether. It also involves ideas about *disincentivizing* behavior by creating deterrents against MEV extraction. The second strategy, tries to reduce the negative effect of MEV extraction by *democratizing* the process by making MEV more accessible to a broader group of network participants. A significant part of their research focuses on so-called L2 chains.

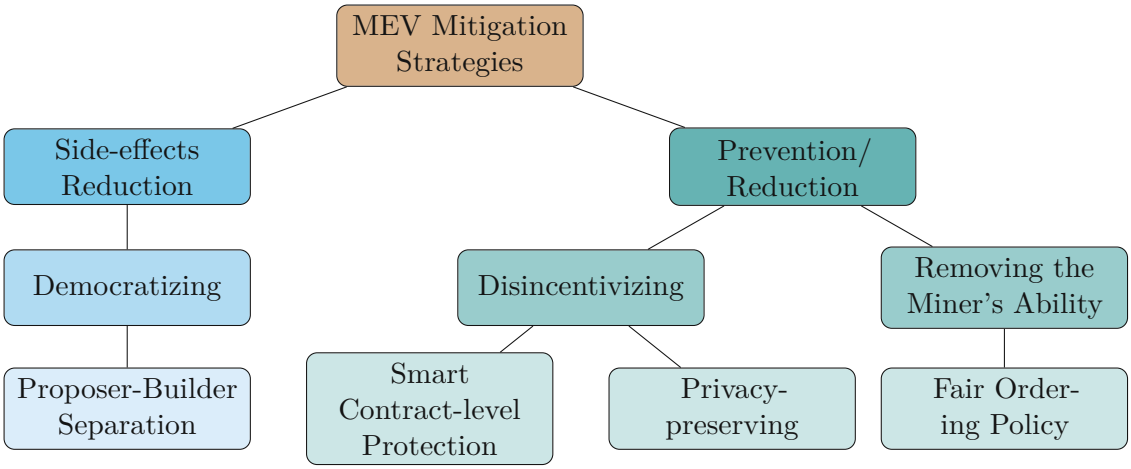

Figure 5.2: Taxonomy of MEV Mitigation Strategies by Alipanahloo et al. [123]

5.3.1 Prevention/Reduction

Fair Ordering Policy

A fair ordering policy can be any method designed to eliminate or limit the ability of a *sequencer* to reorder transactions. Theoretically, this method could significantly reduce the probability of front-running attacks, and sequencers could still censor or delay transactions.

A formal definition of what is considered to be "fair" for a consensus protocol has been proposed by Kelkar et al. [136]. The definition relies on the idea that if a large enough fraction of nodes in a network receives transaction T_1 before T_2, then T_1 should be ordered before T_2.

Fair ordering approaches depend on the protocol's sequencer setup. Therefore, one needs to differentiate between single, multi-, or decentralized sequencers.

A straightforward approach for *single sequencer* setups, which is also used in most notable Rollups, would be implementing a FCFS algorithm. Although this would lead to a fair processing of transactions, it could potentially introduce new attack vectors, such as latency wars and spam attacks. Several solutions have been implemented which have expanded upon the FCFS algorithm (e.g., *TimeBoost* by Mamageishvili et al. [137] which combines timestamps and a bidding system to ensure network stability).

As *single sequencers* suffer from drawbacks, like being a single point of failure regarding trust, *decentralized sequencers* have been proposed as an alternative. These face different challenges, the most prominent being the need to accurately determine transaction order between multiple sequencers and to ensure that each sequencer has the same mempool. This idea is currently being developed by *Metis* [138]. In their approach, all sequencers receive transactions; for each round, a sequencer is chosen to construct a block. The

algorithm for choosing the sequencer is, in essence, round-robin, with all Sequencer Lists stored in a smart contract address controlled by Multiparty Computation (MPC) [139]. Sequencers themselves are included in a permissioned pool, access to which is granted by a Decentralized Autonomous Organization (DAO) through a voting process requiring prospective sequencers to submit proposals. Furthermore, sequencers must stake a certain amount of tokens to deter them from malicious behavior.

Another approach can be found in *Themis* [140], which builds upon the formal definition of fairness. Themis is a leader-based protocol wherein all sequencers transmit their transaction orders to the chosen leader. The protocol uses a fair ordering definition called *Batch-Order-Fairness* stating that if all sequencers receive two transactions T_1 and T_2, and if more than 50% of sequencers received T_1 before T_2, then T_1 is ordered before T_2 by all honest nodes.

Although generally considered to be fair, the fairness of employing a FCFS strategy is considered *"illusory"* by Barczentewicz [116] as it can create an "[...] arms race in speed, similar to the techniques used by high-frequency traders in traditional finance". Participation in such an arms race requires resources, leading back to centralization.

Privacy-Preserving

Contrary to the previous approach, *privacy-preserving* approaches attempt to hide the contents of transactions from the network participants until they are confirmed. One approach in this regard is an *encrypted mempool*. Generally speaking, encrypting happens off-chain when employing encrypted mempools, and decryption happens on-chain.

Three main techniques exist regarding encrypted mempools, which are heavily inspired by dark pools from traditional financial markets [24]:

- **Threshold Encryption** – This method involves multiple trusted parties (key holders or *keypers*) collaboratively decrypting transactions without revealing the decryption key. A symmetric key is divided into shares and distributed among a key-holding committee. Decryption requires a threshold number of shares to reconstruct the key, preventing single-party attacks. Examples include Shutter [141] and F3B [142] protocols.
- **Delay Encryption** – Also known as time-lock encryption, this technique keeps transactions encrypted for a specified duration before decryption. It uses cryptographic primitives like Verifiable Delay Function (VDF) and Time-lock Puzzles to tie decryption to a time-dependent condition. The Radius [143] protocol employs this method in its sequencing layer.
- **Trusted Execution Environment (TEE)** –This approach integrates hardware-based security, typically using Intel's Software Guard Extensions (SGX). Each network node maintains an encryption key within a TEE, where transactions remain encrypted until inclusion in a finalized block. Security relies on the cost associated

with compromising specific hardware devices. An exchange, operating off-chain, which relies on this approach is Tesseract [144].

Each of these approaches comes with a different trust assumption. For Threshold Encryption, trust is placed in the key-holding committee. Employing TEE requires trust in the hardware and the manufacturer. Only delay encryption is trustless, as each user possesses their decryption key, and no committees are necessary.

Apart from mempools, *private transactions*, popularized with the already discussed Flashbots API, are also part of this subcategory of mitigation techniques.

Smart-Contract-Level Protection

Mitigations on a Smart Contract level are designed to limit MEV extraction without requiring changes to the underlying protocol. As we previously mentioned, many dApps are susceptible to MEV extraction, meaning that this approach can not be seen as a comprehensive defense, and usability needs to be considered.

The approaches here can be separated into two main categories regarding where they are being executed: Either they act entirely within the smart contract itself and are therefore *fully on-chain*, or they contain *off-chain* elements.

A classic solution of this category includes the *commit and reveal* approach (sometimes referred to as *content-agnostic ordering* [117] or *blind-order-fairness* [115]). This method consists of two steps:

1. **Commit** – A user sends a transaction with a cryptographic commitment (usually a hash of the value, potentially including a random nonce). [24]

2. **Reveal** – Later, the user reveals the original value and the chosen nonce. This information can be easily verified by the network, binding the revealed data to the previous commit.

However, a significant weakness of this approach is that it is a two-round protocol, with the issue of aborting after the first round becoming problematic. Therefore, such a setup could also be employed maliciously by users sending multiple commit transactions with no intention of executing any of them or only those that later turn out to be profitable or advantageous to them.

An enhanced version of this scheme, named *submarine commitments*, has been devised by Breidenbach et al. [145]. A submarine commit requires the commit transaction to contain a certain amount of non-refundable currency. Afterward, a special address is constructed by hashing user and contract data, thus serving as a commitment. Sending currency to this address seems, to an outside observer, like any other *send to a fresh address*, thereby adding another layer of protection to the commit transaction from being

front-run. For the reveal step, the user then provides a key to the contract, allowing it to reconstruct and verify the commitment address.

Solutions using off-chain elements include *CoWswap* [146] and *FairMM* [147].

CoWswap considers itself an "meta-DEX aggregator" with a *"hybrid MEV protection solution"* containing several key features to address the negative impact of MEV extraction. It does so by setting up a private order flow marketplace where users can swap assets and place limit orders. Traders then sign their intents off-chain, and CoWswap aggregates these in batch settlements on-chain. The mechanism to provide a fair market is called *uniform price clearing*, which ensures that each trader receives the same price for an asset within a block. To offer this consistent price, CoWswap employs a network of solvers who process the orders by competing to find the best execution route across multiple liquidity sources. These solvers are incentivized to compete as they can then capture any resulting surplus profits.

FairMM is a theoretical solution proposed by Ciampi et al. [147], aiming to provide a MEV resistant Market Maker (MM). It works by moving communication between traders and the MM completely off-chain, where traders send trade requests to the MM via secure channels. On the other hand, the MM issues cryptographically signed tickets to the traders, who are then ordered in a queue. The MM then processes the trades on behalf of the traders and posts the details of the trades on-chain.

This idea has been generalized by Heimbach et al. [115], who propose such *professional market makers* as a potential solution. They illustrate this through Hashflow [148], which uses professional market makers to manage liquidity through the Request for Quote (RFQ) model. This approach works with users requesting a quote for their transactions off-chain. A market maker replies with a signed quote (still off-chain), allowing the user to execute the transaction at the offered price. The transaction is then broadcast to the network and, as the price has been agreed upon beforehand, will be executed at the agreed price. Additionally, the Hashflow protocol checks if a better price is available on other DEXs and executes the trade there if so. An apparent weakness of this approach is the possibility of being front-run by the market maker.

5.3.2 Side Effect Reduction

Proposer-Builder Separation

PBS represents a significant upcoming upgrade to the Ethereum network, which aims to mitigate the previously discussed negative impacts of MEV extraction. The fundamental idea is separating the roles of *block proposers* and *block validators*, which should help decentralize power between network participants.

In the current implementation of Ethereum's PoS system, validators serve two roles: they propose blocks and validate them, leading to validators employing complex strategies to maximize their profits. Such incentives could cause centralization of the network in the hands of a few institutional actors. [115]

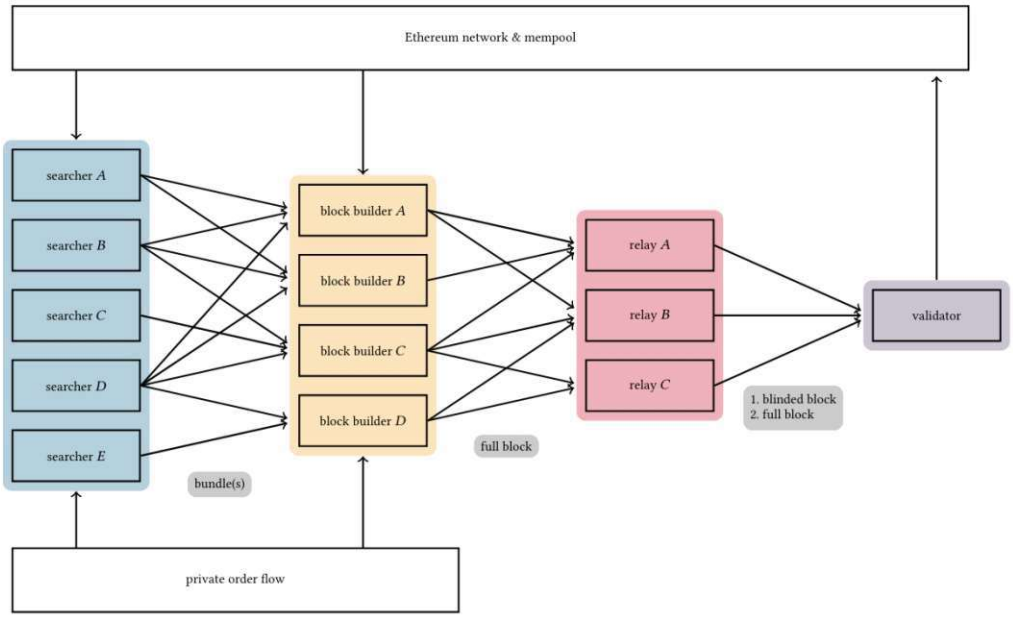

Figure 5.3: Illustration of the PBS scheme by Heimbach et al. [115]

To understand the basic idea of PBS, we need to consider the different roles in the scheme depicted in fig. 5.3:

- **Searchers** – Ethereum users who prioritize privacy and use a private transaction pool instead of the public mempool. Most commonly, these are MEV bots searching for MEV opportunities but also include other participants looking for protection from front-running attacks.

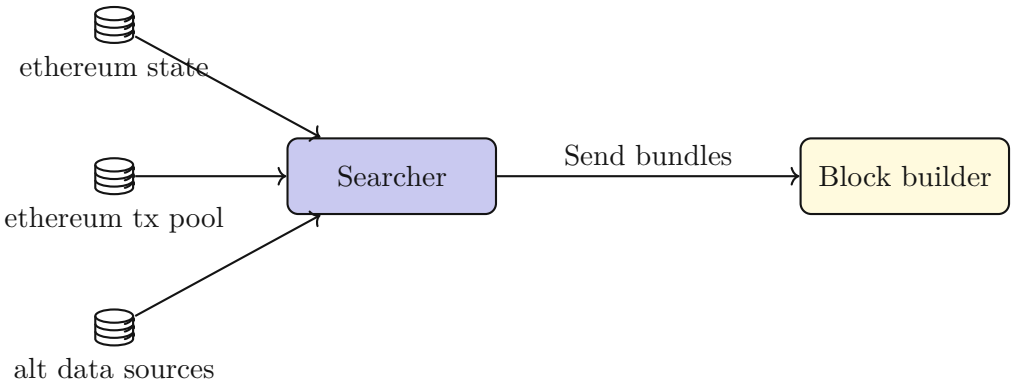

Figure 5.4: Searcher and Block Builder Interaction in PBS

- **Block builders** – Participants who receive immutable *bundles* from searchers. Bundles are one or more transactions (containing private transactions and trans-

actions from the public mempool) grouped and executed in the order they are provided. Block builders then try to create the most profitable block possible.

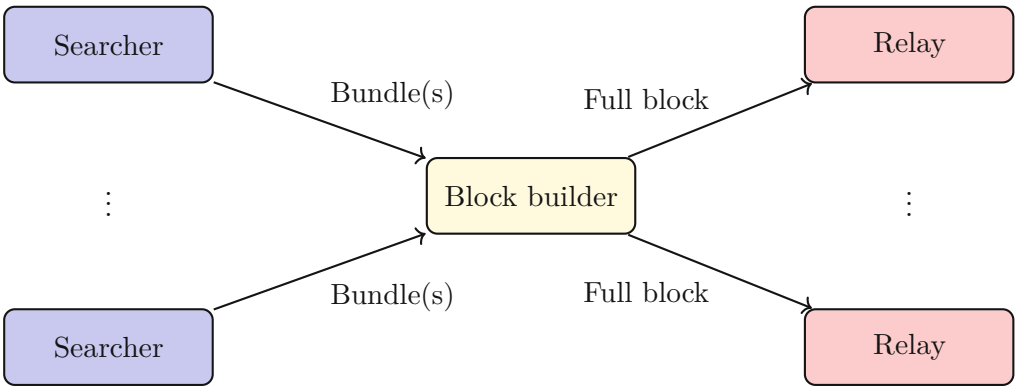

Figure 5.5: Searchers, Block Builder, and Relays Interaction

- **Relays** – Relays hold the blocks from builders in escrow for validators. They accept blocks from builders and send the header of the most profitable block to validators. Only after receiving a signed header from the validators do they release the whole block to the validators.

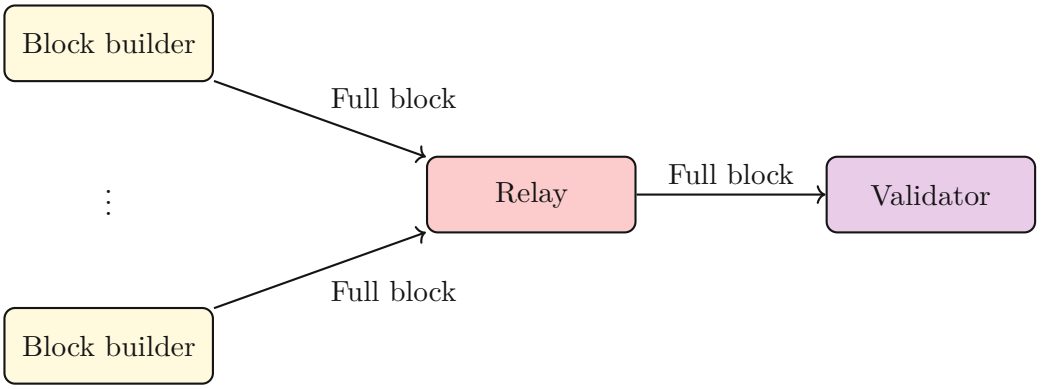

Figure 5.6: Block Builders and Relays Interaction

- **validator** – Finally, validators still need to propose blocks to the network. They can connect with several relays and choose the most profitable block to propose.

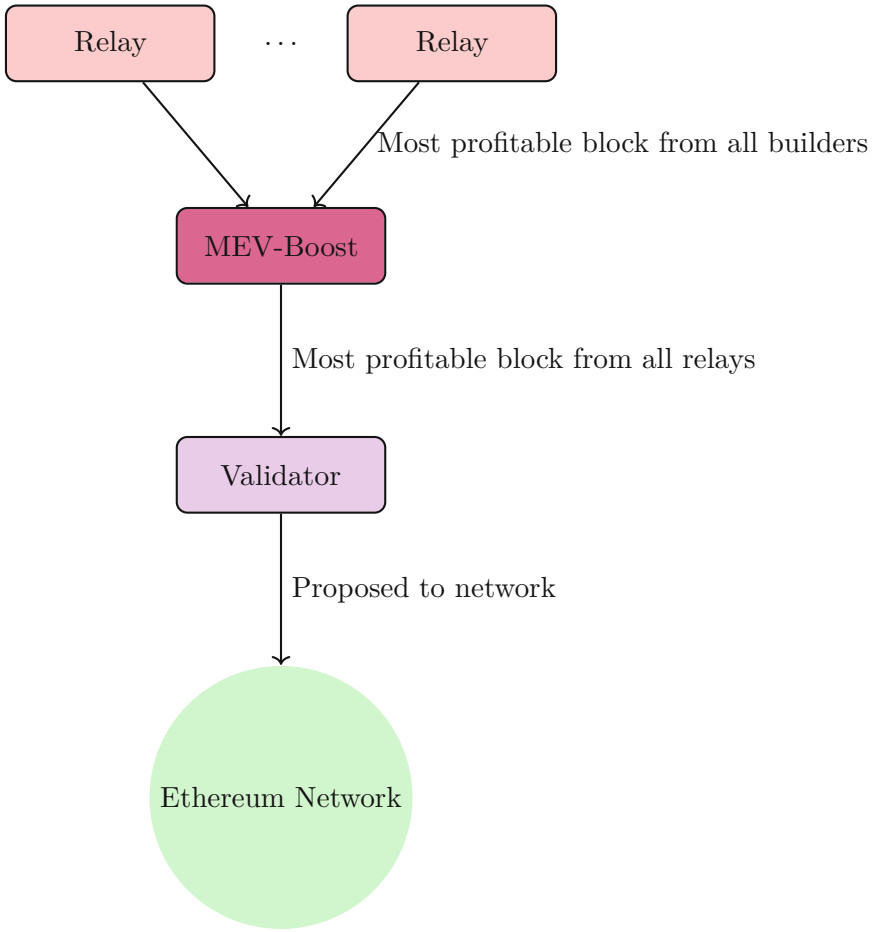

Figure 5.7: MEV-Boost, Validator, and Ethereum Network Interaction

It tackles issues regarding centralization as the economics of MEV extraction are reconfigured. Block proposers no longer do their own MEV searching since they pick a block from many blocks builders offer. This way, even if block builders have performed sophisticated MEV extraction, the reward goes to the proposer. Therefore, even if the MEV extraction market is dominated by a small pool of specialized block builders, the rewards can go to any validator on the network [124].

MEV-Boost

While PBS has not yet been incorporated into the Ethereum network consensus layer at the time of writing, off-chain solutions have emerged. Flashbots, a research and development organization aiming to mitigate the negative impacts of MEV extraction [149], has become a major player. They are the authors of `mev-boost`, an open-source middleware that implements the PBS for Ethereum after the merge and is run by validators [150]. `mev-boost` enjoys a high level of success, with current numbers suggesting that around 90% of blocks since the merge have been produced using it [133].

The basic workflow of `mev-boost` is as follows:

1. Searchers create bundles of transactions coming from either public or private sources and send these to a network of block builders (these transactions are **not** sent to the public mempool).
2. Block builders use these bundles to create the most profitable block possible (this includes both transaction fees and MEV extraction).
3. The constructed block is then submitted to a relay, responsible for checking the block's validity and profitability and holding it in escrow for the validator.
4. Validators select the most profitable block from various relays through `mev-boost`. The relay maintains the privacy of a block's content until the validator proposes the block for inclusion on the main chain.

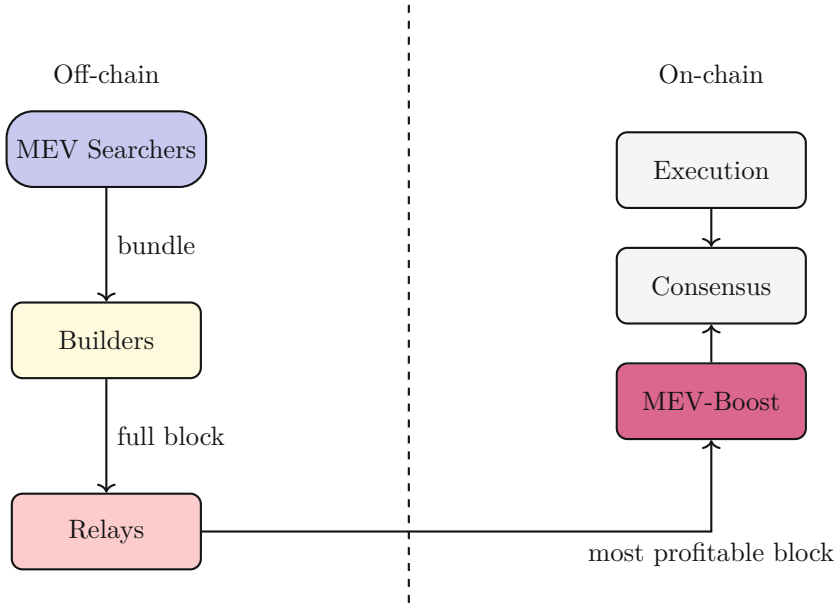

Figure 5.8: MEV-Boost architecture overview [151] [123]

An issue with this scheme is that it is not a complete solution. Firstly, it incentivizes builder centralization, which shifts the need for trusting validators to a different actor. Secondly, it only partially avoids front-running attacks, as the builders can still act as searchers and include their transactions in the block, thus front-run the user. Other risks, such as collusion between builders and relays or the possibility of malicious relays, also remain [52].

Still, the benefits do outweigh the risks, which may be seen as one cause for the rapid adoption of `mev-boost`. Furthermore, it adds no latency to the network as it operates off-chain.

5.3.3 Other Solutions

Various other MEV mitigation solutions exist, which have not been considered in the previous taxonomy. These are briefly discussed in the following sections.

MEV redistribution

A countermeasure to the centralization of proceeds from MEV extraction is to provide mechanisms for redistributing the captured value to the network participants. The basic idea is to open possibilities for users to capture a significant percentage of the MEV created by their transactions. An implementation of this idea is `MEV-Share`, again by Flashbots, which adds a new entity called *matchmaker* to the process [152].

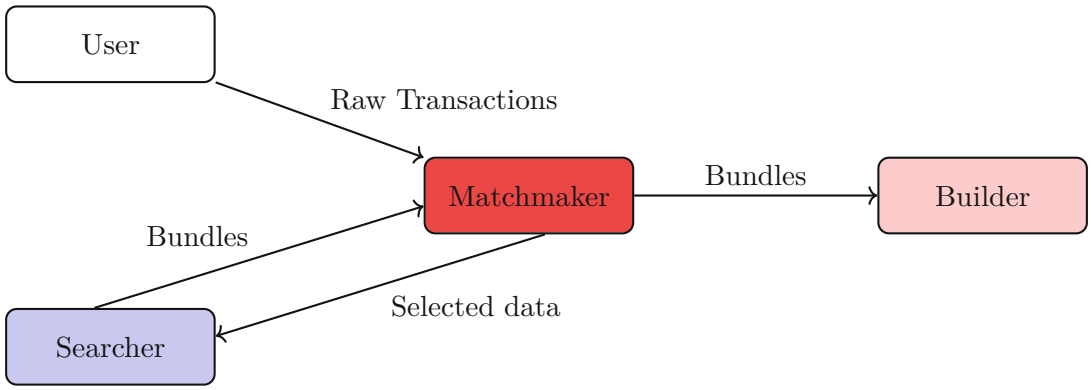

Figure 5.9: MEV-Share architecture [152]

Using `MEV-Share`, users can choose exactly what transaction data should be hidden or revealed through the concept of *programmable privacy*. The matchmaker is responsible for inserting user-submitted transactions into incomplete searcher bundles and simulating their execution, looking for profitable bundles (*matches*). If any matched bundles are discovered, these are sent to builders with the condition that MEV payments are made to the users. `MEV-Share`, therefore, enables collaboration between searchers and users.

eUTXO Model

The Extended UTXO (eUTXO) model, introduced by Cardano, is an extension of the Unspent Transaction Output (UTXO) model, which is the underlying model of Bitcoin. In this model, a user's coins are not accumulated into an account balance but instead stored as individual UTXOs connected to their account. Each transaction takes UTXOs as input, destroys them in the process, and creates new UTXOs as output.

The eUTXO model extends this concept by allowing the introduction of arbitrary logic in the form of scripts. Lanningham [153] proposes a translation from the current Uniswap AMM model into the eUTXO model in the original SundaeSwap whitepaper. In this

translation, a liquidity pool's assets are stored in an eUTXO, effectively simulating a Smart Contract.

Each time a transaction is submitted, it interacts with the eUTXO and creates a new eUTXO as output. Such a setup would definitely remove any possibility of front-running attacks, as a transaction can only be executed in the desired state [115]. However, only one transaction referencing a liquidity pool could be executed per block, as all subsequent transactions would automatically become invalid due to referencing an outdated eUTXO.

5.3.4 Case Studies

Case Study 1: Attack by the Peraire-Bueno brothers

The discussed system, which uses `mev-boost`, PBS, and the new setup of PoS, has many advantages and disadvantages. In a recent indictment by the US court system, significant weaknesses of this setup have been uncovered. The case involves a sophisticated front-running attack by two brothers who were able to extract $25 million in a matter of seconds by exploiting a bug in `mev-boost` and the PBS scheme.

According to the indictment [154], the preparations for the attack began by establishing 16 Ethereum validators (while obfuscating their identities). Then, by deploying several transactions, they were able to reverse-engineer the behavior of MEV bots active in the network. They did so by having their validators publish a set of *"bait" transactions*, which appeared to be lucrative opportunities for MEV extraction [155].

The actual attack then used one of their validator nodes to publish eight transactions, luring in three MEV bots believing to have spotted MEV opportunities.

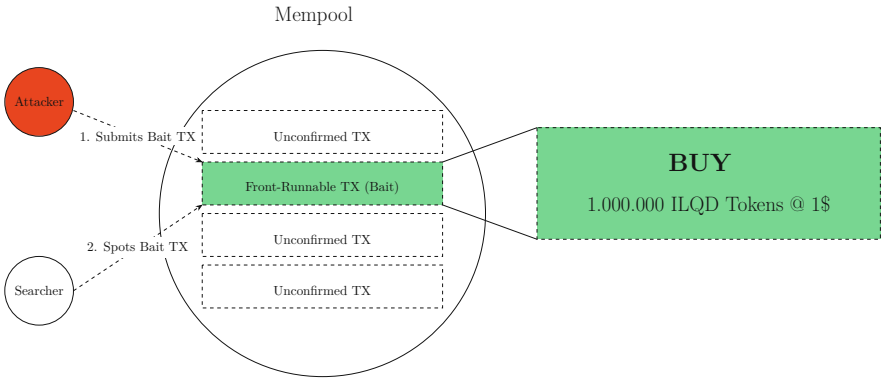

Figure 5.10: First step, attacker publishes bait transaction buying a large amount of illiquid asset which seems like ordinary MEV opportunity to searchers

These bots were, in turn, tricked into proposing transaction bundles containing the bait transactions.

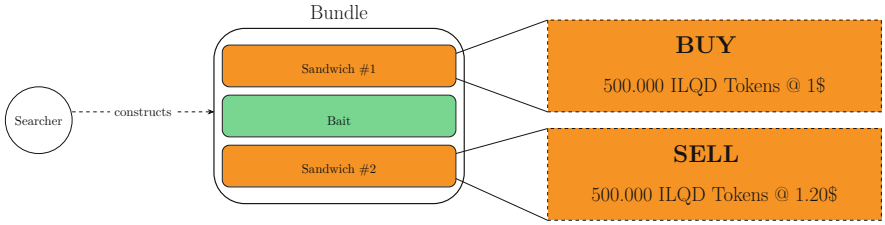

Figure 5.11: Second step, searcher generates bundle for ordinary sandwich attack

Given that the validator nodes were under their control, they produced a fake signature for the relay that proposed the block.

A fundamental aspect of this attack is the vulnerability in the `mev-boost` relay code, giving them access to the entire contents of the blocks. As the relayer was under the impression that the signature was valid, they published the contents of the block, which the attacker's validator used to inspect the contents, reorder the transactions, and then *officially* propose the block with the correct signature.

The original block contained a classic sandwich attack against them. As the attackers published transactions in which they intended to buy a large amount of a highly illiquid asset (pushing up their price in the process), the searcher's original transactions would have front-run them to buy the assets before them and unloaded the funds again before the attackers could do it.

Instead, in the reordered attackers block, they front-run the front-running searchers by buying the assets first, causing the searchers to buy more of the same assets, driving up the prices and finally, the attackers unloading all the assets and thereby draining all the liquidity from the pool.

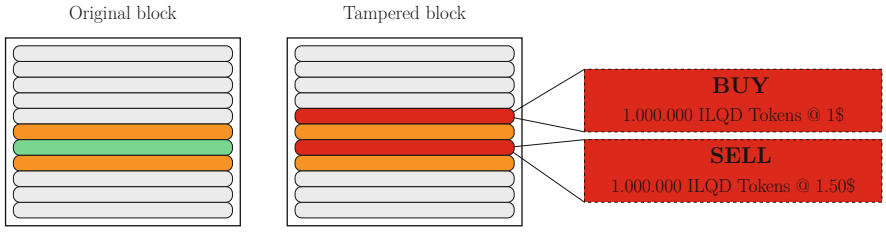

Figure 5.12: Third step, attacker orders the bait transactions in a way that the searchers would front-run the attacker's transactions

This outcome left the searchers with essentially worthless assets that they could not sell and the attackers with highly liquid stablecoins worth millions.

Case Study 2: MEV bots as a force for good

In August 2024, the EVM blockchain Ronin fell victim to an MEV bot extracting $12 million [156]. After a contract upgrade, a bug was introduced, which allowed an attacker

to exploit it and drain the contract's funds. Upon spotting this vulnerability, multiple transactions trying to exploit it were submitted manually. These transactions have been picked up quickly by a MEV bot who front-ran them and was able to withdraw the maximum amount possible in a single transaction. In the aftermath of this exploit, the Ronin team was able to recover the stolen funds by providing a bug bounty to the person responsible for the bot [157].

Interestingly, this incident highlighted a potentially positive aspect of MEV bots acting as inadvertent security measures against malicious actors.

During a similar incident in 2023, an exploit in a compiler led to a large-scale attack against Curve Finance, resulting in losses of approximately $70 million [158]. Once again, MEV bots front-ran the transactions and were able to recover some stolen funds successfully. However, it is worth noting that most other front-run transactions executed by MEV bots were not returned to the victims [159]. This highlights that in some scenarios, the competitive nature of MEV extraction can lead to positive outcomes with the caveat that their primary motivation remains profit extraction.

CHAPTER 6

Conclusion

This chapter serves as a summary of the thesis and a reflection on the findings. It will discuss the results and propose directions for future research.

6.1 Discussing research questions

6.1.1 Definition and detection

The definition of front-running is a challenging undertaking, and no consensus has been reached yet due to the difficulty of defining when an attack occurs and when it does not.

The review presented in chapter 4 attempts to show the variability in definitions we have observed in the literature. The range is wide enough to contain some definitions that do not even consider front-running to be a vulnerability at all but rather a characteristic of the blockchain or a necessary part of the consensus mechanism.

Some authors provide formal definitions with rigorous criteria for what constitutes an instance of front-running considering state inconsistency [33] and transaction races [32]. Interestingly, the financial damage is not considered necessary in all approaches, although others (e.g., [37]) consider it a crucial characteristic and, therefore, include the assets of the participants in the definition.

Some research has introduced new sub-categories of front-running and transaction ordering dependencies such as TOD-Amount, TOD-Ownership, TOD-Transfer, and TOD-Selfdestruct [45] or a distinction between destructive and tolerating front-running [49].

Generally, we encountered several key components of definitions that have been identified in the literature and used in varying combinations for each definition:

1. **Transaction Ordering** - The concept of transactions being executed in different orderings leading to different results.

2. **State changes** - The changes that occur through the execution of transactions.
3. **Storage variables** - Conflicts in reading and writing to storage variables.
4. **Financial damage** - Financial losses to one or more participants.

A universally accepted definition of front-running vulnerabilities has yet to be found, resulting in a situation in which many papers discussing front-running vulnerabilities or adjacent topics lack a definition altogether.

6.1.2 Effects and Exploits

Front-running, MEV extraction, and related concepts affect the smart contract ecosystem in various ways. The complexity of researching this phenomenon stems from the fact that there is no consensus on which practice is considered malicious (or *toxic*) and which is not. Legislative aspects have only been briefly mentioned in this paper, but they could play a defining role in how this consensus could be reached.

Security

Researchers have expressed concerns about the stability of the consensus mechanisms if MEV rewards surpass mining rewards [16], making several attacks, such as time-bandit attacks, possible. On the other hand, some case studies have shown that MEV bots can (albeit inadvertently) act as security measures against malicious actors.

Efficiency

Front-running attempts can quickly lead to increased network congestion and, thus, to higher transaction fees for all participants due to PGAs. However, some researchers argue that MEV bots, especially arbitrage bots, are necessary for stabilizing prices across different exchanges, providing liquidity, and generally improving market efficiency. A trade-off between throughput and network efficiency exists, as solutions (such as the eUTXO model) that eliminate front-running come at the cost of reduced transaction throughput.

Economics

MEV extraction is a zero-sum game, in which extractors can only profit from other participants' losses. Financial damage from front-running, and MEV extraction in general, has been substantial in the past, with the cumulative user loss amounting to $60 million on Ethereum alone for a short timeframe of several months. When observing sums of this magnitude, the question is where these profits flow to, with some indications of a high degree of centralization among very few sophisticated actors. Concepts have been proposed to counteract this, such as `mev-share`, which aims to recapture some profits and distribute them to the network.

Performance

MEV extraction has gained significant attention. The prevalence of this phenomenon led to the development of specialized software such as `mev-boost`, which has become a fundamental component of the Ethereum ecosystem. It is used in about 90% of block production, which has centralized around few active participants, raising concerns about the centralization of the network.

6.1.3 Mitigation

Two primary approaches regarding the mitigation of MEV extraction have been proposed:

1. **Side-effects Reduction** - Strategies aiming to reduce the negative side-effects of MEV extraction, thus taking MEV extraction as an unavoidable part of the ecosystem.
2. **Prevention/Reduction** - Strategies focusing on removing or limiting the ability for reordering transactions, excluding the possibility of MEV extraction.

From the proposed strategies, the more promising approach is side effects reduction, with PBS being the most prominent example. Furthermore, it is expected that in the near future, PBS will become part of the standard Ethereum protocol itself.

6.2 Outlook

It is expected that discussions about the topic of FR, TOD, and MEV will continue and gain importance with blockchain technology finding its way into classical financial markets and subsequent regulatory frameworks. Therefore, it is important to investigate this phenomenon, define clear terminology, specify characteristics, and define which behavior is desirable and which is not. Currently, many aspects need to be better defined and are subject to ongoing research. This comes at the expense of a large part of the user base and hampers further adoption. Therefore, if blockchain technology is to become a fundamental aspect of global finance, aspects like these need to be resolved.

APPENDIX A

Methodology for the analysis

A.1 Data collection

Data has been collected from approximately 1000000 blocks starting from block height 19475310 to 20475310. Two sources were considered for the analysis which expose API endpoints to access MEV data:

- `https://zeromev.org/` - A website tracking MEV extraction in real-time and providing data about MEV transactions for each block.
- `https://libmev.com/` - A website providing data about MEV searchers and various statistics about their activity on the Ethereum network.

A.1.1 Data from `https://zeromev.org/`

The endpoint provided by `https://zeromev.org/` is queried using a python script and the resulting data is stored as JSON file for each block.

Exemplary result for block 19475353:


```
    [
        {
          "block_number": 19475353,
          "tx_index": 2,
          "mev_type": "sandwich",
          "protocol": "uniswap2",
          "user_loss_usd": -32.47,
          "extractor_profit_usd": null,
          "user_swap_volume_usd": 3246.6,
          "user_swap_count": 1,
          "extractor_swap_volume_usd": null,
          "extractor_swap_count": null,
```

```
        "imbalance": null,
        "address_from":
↪ "0x2f8a9f358e7aa1d5e24e0f812d7cdac1d5dcdafc",
        "address_to": "0x3fc91a3afd70395cd496c647d5a6cc9d4b2b7fad",
        "arrival_time_us": "2024-03-20T10:36:27.755592+00:00",
        "arrival_time_eu": "2024-03-20T10:36:26.834326+00:00",
        "arrival_time_as": "2024-03-20T10:36:28.051844+00:00"
      },
      {
        "block_number": 19475353,
        "tx_index": 3,
        "mev_type": "backrun",
        "protocol": "uniswap2",
        "user_loss_usd": null,
        "extractor_profit_usd": null,
        "user_swap_volume_usd": null,
        "user_swap_count": null,
        "extractor_swap_volume_usd": 1456.21,
        "extractor_swap_count": 1,
        "imbalance": null,
        "address_from":
↪ "0xae2fc483527b8ef99eb5d9b44875f005ba1fae13",
        "address_to": "0x6b75d8af000000e20b7a7ddf000ba900b4009a80",
        "arrival_time_us": "2024-03-20T10:36:40.899073+00:00",
        "arrival_time_eu": "2024-03-20T10:36:38.874455+00:00",
        "arrival_time_as": "2024-03-20T10:36:42.01999+00:00"
      },
      ... # Further transactions in the block
    ]
```

A python script was then written to query the endpoint and store the results in a CSV file for further analysis.

```python
import json
import requests
import pandas as pd

API_URL = "https://data.zeromev.org/v1"
ENDPOINT = "mevBlock"

# TIMESTAMPS
BEGINNING = "1710930455"  # 20.3.2024
END = "1723016999"  # 7.8.2024

# BLOCKS
FIRST_BLOCK = 19475310
LAST_BLOCK = 20475310

def get_mev_block_data(block_number: int, num_blocks: int = 1):
    """
    Returns all related txs for the passed block_number, or
    the number of blocks specified by count, starting from the block_number
    """

    if num_blocks > 100:
        raise ValueError("Value must be less than 100")

    url = f"{API_URL}/{ENDPOINT}?block_number={block_number}&count={num_blocks}"

    response = requests.get(url)
    data = response.json()
    with open("res.json", "w") as f:
        json.dump(data, f, indent=4)
    return data


def get_all_blocks(batch_size: int = 1, output_file: str = "all_blocks_data.csv"):
    """
    Get all blocks from the API and save them to a CSV file.

    Args:
        batch_size (int): Number of blocks to fetch at a time.
        output_file (str): Name of the CSV file to save the data to.
    """
    all_data = pd.DataFrame()
    for block_number in range(FIRST_BLOCK, LAST_BLOCK):
        data = get_mev_block_data(block_number, num_blocks=batch_size)
        df = pd.json_normalize(data)
        all_data = pd.concat([all_data, df], ignore_index=True)

    all_data.to_csv(output_file, index=False)
```

Plotting the cumulative user loss for all blocks in the dataset for figure 5.1:

```python
import pandas as pd
import matplotlib.pyplot as plt

def cumulative_usd_loss_plot(
    path_to_csv: str, output_path: str = "figures/cumulative_user_loss.png"
):

    df = pd.read_csv(path_to_csv)

    df["cumulative_user_loss"] = df["user_loss_usd"].cumsum()

    # Create the bar plot
    plt.figure(figsize=(12, 6))
    plt.plot(df["block_number"], df["cumulative_user_loss"])

    plt.title("Cumulative User Loss vs Block Number")
    plt.xlabel("Block Number")
    plt.ylabel("Cumulative User Loss (USD)")

    # Format y-axis labels to show in millions
    plt.gca().yaxis.set_major_formatter(
        plt.FuncFormatter(lambda x, p: f"${x/1e6:.1f}M")
    )

    # Format x-axis labels to show full block number in steps of 50000
    plt.gca().xaxis.set_major_locator(plt.MaxNLocator(10))
    plt.gca().xaxis.set_major_formatter(plt.FuncFormatter(lambda x, p: f"{int(x)}"))

    plt.xticks(rotation=45)

    plt.tight_layout()

    # Uncomment to save plot
    # if output_path is not None:
    #     plt.savefig(output_path)

    # Uncomment to show plot
    # plt.show()
```

Calculating the distribution of losses for table 5.1:

```
    def loss_distribution(path_to_csv: str, output_path: str = "
    loss_distribution.csv"):
        data = pd.read_csv(path_to_csv)
        loss_categories = [(0, 50), (51, 100), (101, 500),
            (501, 1000), (1001, 5000), (5001, 10000), (10001, float("inf"))]

    # Create labels for the categories
    labels = ["$0-$50", "$51-$100", "$101-$500", "$501-$1000", "$1001-$5000",
     "$5001-$10000", "$10000+"]

    # Categorize the losses
    data["loss_category"] = pd.cut(
        data["user_loss_usd"],
        bins=[c[0] for c in loss_categories] + [float("inf")],
        labels=labels,
    )

    # Create new df for loss distribution
    loss_distribution = pd.DataFrame(data["loss_category"].value_counts().
    sort_index())

    loss_distribution["total_loss_usd"] = data.groupby("loss_category",
    observed=True)[
        "user_loss_usd"
    ].sum()

    loss_distribution["percentage"] = round(
        100
        * loss_distribution["total_loss_usd"]
        / loss_distribution["total_loss_usd"].sum(),
        2,
    )

    loss_distribution["percentage_of_counts"] = round(
        100 * loss_distribution["count"] / loss_distribution["count"].sum(),
        2,
    )

    # sanity  check that the percentages sum to 100
    assert loss_distribution["percentage"].sum() == 100

    loss_distribution.to_csv(output_path)
```

A.1.2 Data from `https://libmev.com/`

libMEV provides information about MEV searchers and their profits in so-called leaderboards. These can further be filtered by tags, such as sandwich attacks and a timeframe.

A python script was written to query the API and store the results in a CSV file for further analysis. Results are presented in section B.1.

```
1
2  # TIMESTAMPS
3  BEGINNING = "1710930455"  # 20.3.2024
4  END = "1723016999"  # 7.8.2024
5
6  def lib_mev_stats_by_tag(tag: str):
7      URL = f"https://api.libmev.com/v1/searchers/leaderboardSummaries?
       timestampRange={BEGINNING},{END}&filterByTags={tag}&offset=0&orderByDesc=
       profit_usdc"
8
9      response = requests.get(URL)
10     data = response.json()
11     data = data["data"]
12     df = pd.json_normalize(data)
13
14     # round some columns to 2 decimal places
15     df["profit_usdc"] = df["profit_usdc"].apply(lambda x: round(x, 2))
16     df["profit_eth"] = df["profit_eth"].apply(lambda x: round(x, 2))
17
18     # convert profit_margin to percentage
19     df["profit_margin"] = df["profit_margin"].apply(lambda x: round(x * 100,
       2))
20
21     df.to_csv(f"stats/lib_mev_{tag}.csv", index=False)
```

APPENDIX B

MEV Sandwich attacks

B.1 Sandwich attackers and their profit

Table B.1: Top 50 MEV Sandwich Attackers by Profit

	Profit Margin	Profit (USDC)	Bundle Count	Searcher Contract
1	36.42%	1,489,041.03	6,779	0x00000000003b3cc22af3ae1eac0440bcee416b40
2	24.66%	1,131,613.02	78,440	0x000000d40b595b94918a28b27d1e2c66f43a51d3
3	1.49%	873,282.33	458,065	0x6b75d8af000000e20b7a7ddf000ba900b4009a80
4	40.05%	341,778.68	1,211	0x000000000035b5e5ad9019092c665357240f594e
5	15.74%	166,935.5	11,884	0x00000000009e50a7ddb7a7b0e2ee6604fd120e49
6	7.57%	160,423.3	4,659	0xe8c060f8052e07423f71d445277c61ac5138a2e5
7	3.91%	61,544.00	16,931	0x429cf888dae41d589d57f6dc685707bec755fe63
8	20.59%	36,326.59	1,067	0xb0000000aa4f00af1200c8b2befb6300853f0069
9	1.37%	26,325.85	85	0x64545160d28fd0e309277c02d6d73b3923cc4bfa
10	10.73%	22,819.86	2,038	0x3e0000de5060979f9b4e002e81ab000000b60093
11	0.71%	21,079.60	53,232	0x00000000a991c429ee2ec6df19d40fe0c80088b8
12	16.40%	14,637.78	1,957	0x000000000c1500d6cf5a65167f131a53c82c1033
13	0.65%	6,568.83	22,820	0x4736b02db015dcd1a57a69c889d073b100000000
14	1.81%	6,484.13	6,953	0xdc3bfcb3521cd02af66d394f817a0e6fe62ded95
15	49.66%	6,248.35	29	0x3face0004bc1003b9d0672e7b23134cb80115fb6
16	6.34%	5,719.86	590	0x2ecd8f51fa432415fe8f385b8b369f13eff16fcb
17	0.90%	3,883.56	4,957	0x93ffb15d1fa91e0c320d058f00ee97f9e3c50096
18	21.82%	3,801.82	26	0xe545c3cd397be0243475af52bcff8c64e9ead5d7
19	7.01%	3,126.39	554	0x0000a42df58060230d7f1aefe47da338078244e8
20	0.18%	2,460.05	18	0x6980a47bee930a4584b09ee79ebe46484fbdbdd0
21	1.30%	2,122.34	7	0xe08d97e151473a848c3d9ca3f323cb720472d015
22	10.45%	1,557.50	211	0x3c005ba2000f0000ba000d69000ac8ec003800bc
23	8.53%	1,533.92	554	0x00df657aa9a100a600001700004a00359a639f47
24	7.80%	1,185.33	366	0x0000a89d6d8f00f90ada0000003e005eaa4df87f
25	21.10%	760.12	1	0x525145f821d8d2abb494c454e9445e14c817f7cb
26	36.64%	643.07	3	0x53facee52e897740b140f5304e9cd9dc6238d735
27	0.73%	446.46	574	0xcc2687c14915fd68226ccf388842515739a739bd
28	2.00%	338.01	130	0x0000000000c2f3017e5af636ea91bd68ec3888ed
29	10.35%	334.88	2	0x0000000000a84d1a9b0063a910315c7ffa9cd248
30	3.20%	320.49	11	0x8d27c03fa77f30af3dce552dde5babf65d14861f
31	16.77%	253.17	2	0xdea6fdea0471ce1545331b7b93fdbd43786fa4c2
32	78.49%	220.62	1	0x95c0c8a41caaa33f22ff042dc4279ed2b4e74041
33	35.52%	159.82	1	0xe4000004000bd8006e00720000d27d1fa000d43e
34	9.30%	73.81	7	0x24902aa0cf0000a08c0ea0b003b0c0bf600000e0
35	2.69%	49.52	47	0x0075006900ce0028c14700306f0000473ec07cc7
36	40.87%	11.40	1	0xe30062750007002400ba00c47e004a0600e500fb
37	39.48%	7.69	1	0x493f461aead031cee2027f1b95370a692611acb9
38	1.42%	4.74	2	0x0ddc6f9ce13b985dfd730b8048014b342d1b54f7
39	38.24%	3.24	2	0x0eae044f00b0af300500f090ea00027097d03000
40	-1.38%	-465.33	351	0x76b5a83c8c8097e7723eda897537b6345789b229
41	-98.21%	-574.85	5	0x6f1cdbbb4d53d226cf4b917bf768b94acbab6168
42	-0.88%	-865.29	5	0x7d32c90762e22379235fc311fdb16fab399ed40a
43	-249.20%	-1626.04	13	0x0b69657070571f8f9987cef93b77faea852f51b8
44	-21.70%	-2658.73	17	0x5ed5dd65ab0dc1bccc44eedaa40680c231faaa9f
45	-9.27%	-3314.05	2314	0x9062db3da0bf1e83acdbe5e976668c87f8fef5e1
46	-11.32%	-3545.97	488	0x000000000000521fa5ddd7611a38f9dc280cd2f2
47	-45.05%	-13549.53	391	0x5050e08626c499411b5d0e0b5af0e83d3fd82edf
48	-433.27%	-37549.65	35	0x1fdb319cc1be16ff75ef84e408b0bc1594dd4d3c
49	-839.89%	-47799.07	43	0x00a2712e3200e89c6b8500b2da5c6c9431330000
50	-18.99%	-191924.15	13415	0xa69babef1ca67a37ffaf7a485dfff3382056e78c

Table B.2: Top 50 MEV Sandwich Attackers by Profit (USDC) between blocks 19475310 and 20475310

Overview of Generative AI Tools Used

During the work on this thesis, the following AI tools were used:

Grammarly

Grammarly[1] is a popular tool for proofreading, checking grammar and punctuation, as well as improving writing style.

Usage

Grammarly has been used in the entire document to improve legibility and ensure correctness.

Perplexity

Perplexity[2] is an AI-enhanced search engine that is capable of answering questions and providing information by accessing various sources on the internet.

Usage

Perplexity has mainly been used to quickly retrieve a vast array of different sources on a given topic. In particular, it was used to identify possible starting points that can be explored, for retrieving sources when researching a specific topic, as well as checking the own contribution for plausibility, completeness, and correctness.

The generated answers have at no point been used verbatim as part of the thesis but rather as a source of information to be further processed by the author.

[1] https://www.grammarly.com/
[2] https://www.perplexity.ai/

List of Figures

List of Tables

Listings

Acronyms

AMM Automated Market Maker. 11, 36, 51

CA Contract Account. 8, 9

CBOE Chicago Board Options Exchange. 13

CWE Common Weakness Enumeration. 25

DAO Decentralized Autonomous Organization. 44

dApp Decentralized Application. 1, 8, 45

DeFi Decentralized Finance. 2, 37, 38

DEX Decentralized Exchange. 10, 11, 17, 22, 46

EOA Externally Owned Account. 8, 9

eUTXO Extended UTXO. 51, 52, 56

EVM Ethereum Virtual Machine. 1, 8, 28, 30, 31, 53

FCFS First-Come-First-Serve. 38, 43, 44

FR Front Running. 2, 31, 33, 35, 36, 57

MEV Maximal Extractable Value. 4, 35–39, 41, 42, 46, 47, 49–54, 56, 57, 59, 64, 67

MM Market Maker. 46

MPC Multiparty Computation. 44

PBS Proposer/Builder Separation. 41, 42, 46, 47, 49, 52, 57

PGA Priority Gas Auction. 38, 56

PoS Proof of Stake. 7, 41, 46, 52

PoW Proof of Work. 7, 41

RFQ Request for Quote. 46

SEC Securities and Exchange Commission. 13

SWC Smart Contract Weakness Classification. 25

TEE Trusted Execution Environment. 44, 45

TOD Transaction Order Dependence. 14, 15, 25, 28–34, 57

UTXO Unspent Transaction Output. 51

VDF Verifiable Delay Function. 44

Bibliography

[1] V. Buterin, "Ethereum white paper: A next generation smart contract & decentralized application platform," 2013. [https://ethereum.org/en/whitepaper/ accessed on 16. Oct. 2023].

[2] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," May 2009. [http://www.bitcoin.org/bitcoin.pdf accessed on 16. Oct. 2023].

[3] A. T. Sherman, F. Javani, H. Zhang, and E. Golaszewski, "On the origins and variations of blockchain technologies," *IEEE Security and Privacy*, vol. 17, no. 1, pp. 72–77, 2019. https://doi.org/10.1109/MSEC.2019.2893730.

[4] C. Team, "2022 Biggest Year Ever For Crypto Hacking - Chainalysis," *Chainalysis*, Aug. 2023. [https://www.chainalysis.com/blog/2022-biggest-year-ever-for-crypto-hacking accessed 16. Oct. 2023].

[5] C. Diligence, "Frontrunning - Ethereum Smart Contract Best Practices," June 2023. [https://consensys.github.io/smart-contract-best-practices/attacks/frontrunning; accessed 13. Jul. 2023].

[6] K. Christidis and M. Devetsikiotis, "Blockchains and smart contracts for the internet of things," *IEEE Access*, vol. 4, pp. 2292–2303, 2016. https://doi.org/10.1109/ACCESS.2016.2566339.

[7] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," 2014. https://ethereum.github.io/yellowpaper/paper.pdf. Accesssed 10.1.2024.

[8] K. Sultan, U. Ruhi, and R. Lakhani, "Conceptualizing blockchains: Characteristics & applications," *arXiv preprint arXiv:1806.03693*, 2018. https://arxiv.org/abs/1806.03693.

[9] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "An overview of blockchain technology: Architecture, consensus, and future trends," in *2017 IEEE International Congress on Big Data (BigData Congress)*, pp. 557–564, 2017. https://doi.org/10.1109/BigDataCongress.2017.85.

[10] G. Fanti and P. Viswanath, "Deanonymization in the Bitcoin P2P Network," in *Advances in Neural Information Processing Systems* (I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, eds.), vol. 30, p. 1364–1373, Curran Associates, Inc., 2017. ISBN 9781510860964.

[11] A. Biryukov and S. Tikhomirov, "Deanonymization and linkability of cryptocurrency transactions based on network analysis," in *2019 IEEE European Symposium on Security and Privacy (EuroSP)*, pp. 172–184, 2019. https://doi.org/10.1109/EuroSP.2019.00022.

[12] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, pp. 203–226, New York, NY, USA: ACM, Oct. 2019. https://doi.org/10.1145/3335772.3335936.

[13] "The Merge | ethereum.org," June 2024. [https://ethereum.org/en/roadmap/merge accessed 17. Jul. 2024].

[14] V. Buterin, D. Hernandez, T. Kamphefner, K. Pham, Z. Qiao, D. Ryan, J. Sin, Y. Wang, and Y. X. Zhang, "Combining GHOST and Casper," *arXiv*, Mar. 2020. https://doi.org/10.48550/arXiv.2003.03052.

[15] V. Buterin *et al.*, "A next-generation smart contract and decentralized application platform," *white paper*, vol. 3, no. 37, pp. 2–1, 2014. [https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf accessed 17. Jul. 2024].

[16] 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. https://doi.org/10.1109/SP40000.2020.00040.

[17] etherscan. io, "Ethereum Average Gas Price Chart | Etherscan," July 2024. [https://etherscan.io/chart/gasprice accessed 17. Jul. 2024].

[18] B. Charoenwong and M. Bernardi, "Lessons from a decade of cryptocurrency hacks, 2011-2021," in *The Elgar Companion to Decentralized Finance, Digital Assets, and Blockchain Technologies*, pp. 147–166, Edward Elgar Publishing, 2024. https://doi.org/10.4337/9781035307760.00013.

[19] "Top Cryptocurrency Exchanges Ranked By Volume | CoinMarketCap," July 2024. [https://coinmarketcap.com/rankings/exchanges accessed 17. Jul. 2024].

[20] "Top Cryptocurrency Decentralized Exchanges Ranked | CoinMarketCap," July 2024. [https://coinmarketcap.com/rankings/exchanges/dex/?type=spot accessed 17. Jul. 2024].

[21] "DefiLlama," July 2024. [https://defillama.com/dexs accessed 17. Jul. 2024].

[22] J. W. Markham, "'Front-Running' - Insider Trading Under the Commodity Exchange Act," 1988. https://papers.ssrn.com/sol3/papers.cfm?abstract_id=1701942.

[23] D. Muhs, "SWC-114 - Smart Contract Weakness Classification (SWC)," Aug. 2023. [Online; accessed 14. Aug. 2023].

[24] S. Eskandari, S. Moosavi, and J. Clark, "Sok: Transparent dishonesty: front-running attacks on blockchain," in *Financial Cryptography and Data Security: FC 2019 International Workshops, VOTING and WTSC, St. Kitts, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23*, pp. 170–189, Springer, 2020. https://doi.org/10.1007/978-3-030-43725-1_13.

[25] W. Zhang, L. Wei, S.-C. Cheung, Y. Liu, S. Li, L. Liu, and M. R. Lyu, "Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation," *IEEE Transactions on Software Engineering*, 2023. https://doi.org/10.1109/TSE.2023.3270117.

[26] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," pp. 906–917, Oct. 2012. https://doi.org/10.1145/2382196.2382292.

[27] C. F. Torres, R. Camino, and R. State, "Frontrunner jones and the raiders of the dark forest: An empirical study of frontrunning on the ethereum blockchain," in *30th USENIX Security Symposium (USENIX Security 21)*, pp. 1343–1359, USENIX Association, Aug. 2021. https://www.usenix.org/conference/usenixsecurity21/presentation/torres.

[28] "ERC20.sol at master · OpenZeppelin/openzeppelin-contracts," Aug. 2024. [https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol; accessed 9. Aug. 2024].

[29] D. Muhs, "Approval Vulnerabilities - Smart Contract Security Field Guide," July 2024. [https://scsfg.io/hackers/approvals; accessed 9. Aug. 2024].

[30] D. Muhs, "Smart Contract Weakness Classification (SWC)," Aug. 2023. [Online; accessed 14. Aug. 2023].

[31] SmartContractSecurity, "SWC-registry GitHub Repository," Mar. 2024. [Online; accessed 25. Mar. 2024].

[32] C. Ma, W. Song, and J. Huang, "TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts," in *ESEC/FSE 2023: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium*

on the Foundations of Software Engineering, pp. 947–959, New York, NY, USA: ACM, Nov. 2023. https://doi.org/10.1145/3611643.3616281.

[33] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 161–178, May 2022. ISSN: 2375-1207 https://doi.org/10.1109/SP46214.2022.9833721.

[34] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *ACM Conferences*, pp. 363–373, New York, NY, USA: ACM, July 2019. https://doi.org/10.1145/3293882.3330560.

[35] "mythril/mythril/analysis/module/modules/transaction_order_dependence.py at cfa8a1fa58f589ba26ce09b2f1d35a63d7e5c5ec · Consensys/mythril," July 2024. [https://github.com/Consensys/mythril/blob/cfa8a1fa58f589ba26ce09b2f1d35a63d7e5c5ec/mythril/analysis/module/modules/transaction_order_dependence.py accessed 2. Jul. 2024].

[36] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 67–82, New York, NY, USA: ACM, Oct. 2018. https://doi.org/10.1145/3243734.3243780.

[37] W. Zhang, Z. Zhang, Q. Shi, L. Liu, L. Wei, Y. Liu, X. Zhang, and S.-C. Cheung, "Nyx: Detecting exploitable front-running vulnerabilities in smart contracts," in *2024 IEEE Symposium on Security and Privacy (SP)*, pp. 146–146, IEEE Computer Society, 2024. https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00146.

[38] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pp. 254–269, New York, NY, USA: ACM, Oct. 2016. https://doi.org/10.1145/2976749.2978309.

[39] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts," *IEEE Trans. Network Sci. Eng.*, vol. 8, pp. 1133–1144, Jan. 2020. https://doi.org/10.1109/TNSE.2020.2968505.

[40] Y. Li, "Finding concurrency exploits on smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 144–146, IEEE Press, May 2019. https://doi.org/10.1109/ICSE-Companion.2019.00061.

[41] M. Fu, L. Wu, Z. Hong, F. Zhu, H. Sun, and W. Feng, "A critical-path-coverage-based vulnerability detection method for smart contracts," *IEEE Access*, vol. 7,

pp. 147327–147344, 2019. https://doi.org/10.1109/ACCESS.2019.2947146.

[42] Y. Yao, H. Li, X. Yang, and Y. Le, "An improved vulnerability detection system of smart contracts based on symbolic execution," in *2022 IEEE International Conference on Big Data (Big Data)*, pp. 3225–3234, 2022. https://doi.org/10.1109/BigData55660.2022.10020730.

[43] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts.," in *Network and Distributed Systems Security (NDSS) Symposium 2018*, pp. 1–12, 2018. https://doi.org/10.14722/ndss.2018.23082, ISBN 1-1891562-49-5.

[44] Z. Li, S. Lu, R. Zhang, Z. Zhao, R. Liang, R. Xue, W. Li, F. Zhang, and S. Gao, "VulHunter: Hunting Vulnerable Smart Contracts at EVM Bytecode-Level via Multiple Instance Learning," *IEEE Trans. Software Eng.*, vol. 49, pp. 4886–4916, Sept. 2023. https://doi.org/10.1109/TSE.2023.3317209.

[45] S. Munir and C. Reichenbach, "TODLER: A Transaction Ordering Dependency anaLyzER - for Ethereum Smart Contracts," in *2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, p. 14, IEEE. https://doi.org/10.1109/WETSEB59161.2023.00007.

[46] "Ethereum Improvement Proposals," June 2024. [Online; accessed 5. Jun. 2024], https://eips.ethereum.org/EIPS/eip-6780.

[47] G. Ramakrishnan, M. Rehan, and G. Sujatha, "Solidity Vulnerability Scanner," in *2022 International Conference on Data Science, Agents & Artificial Intelligence (ICDSAAI)*, pp. 08–10, IEEE. https://doi.org/10.1109/ICDSAAI55433.2022.10028877.

[48] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in Ethereum smart contracts," *Proc. ACM Program. Lang.*, vol. 3, pp. 1–29, Oct. 2019. https://doi.org/10.1145/3360615.

[49] K. Qin, L. Zhou, and A. Gervais, "Quantifying Blockchain Extractable Value: How dark is the forest?," in *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 22–26, IEEE. https://doi.org/10.1109/SP46214.2022.9833734.

[50] D. Yuan, X. Wang, Y. Li, and T. Zhang, "Optimizing smart contract vulnerability detection via multi-modality code and entropy embedding," *Journal of Systems and Software*, vol. 202, p. 111699, Aug. 2023. https://doi.org/10.1016/j.jss.2023.111699.

[51] X. Tang, K. Zhou, J. Cheng, H. Li, and Y. Yuan, "The Vulnerabilities in Smart Contracts: A Survey," *Commun. Comput. Info. Sci.*, vol. 1424, pp. 177–190, 2024. https://doi.org/10.1007/978-3-030-78621-2_14.

[52] S. Ramos and J. Ellul, "The MEV Saga: Can Regulation Illuminate the Dark Forest?," *Lect. Notes Bus. Inf. Process.*, vol. 482, pp. 186–196, 2024. https://doi.org/10.1007/978-3-031-34985-0_19.

[53] S. S. Kushwaha, S. Joshi, D. Singh, M. Kaur, and H.-N. Lee, "Systematic Review of Security Vulnerabilities in Ethereum Blockchain Smart Contract," *IEEE Access*, vol. 10, pp. 6605–6621, 2024. https://doi.org/10.1109/ACCESS.2021.3140091.

[54] C. Sendner, H. Chen, H. Fereidooni, L. Petzi, J. König, J. Stang, A. Dmitrienko, A.-R. Sadeghi, and F. Koushanfar, *Smarter Contracts: Detecting Vulnerabilities in Smart Contracts with Deep Transfer Learning*. 2023. https://doi.org/10.14722/ndss.2023.23263, ISBN 1-891562-83-5.

[55] X. You, H. Li, H. Wang, and F. Mehmood, "SmartDT: An Effective Vulnerability Detection System of Smart Contracts Based on Deep Learning," in *2023 IEEE International Conference on Big Data (BigData)*, pp. 15–18, IEEE. https://doi.org/10.1109/BigData59044.2023.10386771.

[56] S. Sayeed, H. Marco-Gisbert, and T. Caira, "Smart Contract: Attacks and Protections," *IEEE Access*, vol. 8, pp. 24416–24427, Jan. 2020. https://doi.org/10.1109/ACCESS.2020.2970495.

[57] Q. Zeng, J. He, G. Zhao, S. Li, J. Yang, H. Tang, and H. Luo, "EtherGIS: A Vulnerability Detection Framework for Ethereum Smart Contracts Based on Graph Learning Features," in *2022 IEEE 46th Annual Computers, Software, and Applications Conference (COMPSAC)*, pp. 2022–01, IEEE. https://doi.org/10.1109/COMPSAC54236.2022.00277.

[58] X. Wang, J. Sun, C. Hu, P. Yu, B. Zhang, and D. Hou, "EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection," *Wireless Commun. Mobile Comput.*, vol. 2022, p. 1565007, Jan. 2022. https://doi.org/10.1155/2022/1565007.

[59] O. Lutz, H. Chen, H. Fereidooni, C. Sendner, A. Dmitrienko, A. R. Sadeghi, and F. Koushanfar, "Escort: ethereum smart contracts vulnerability detection using deep neural network and transfer learning," *arXiv preprint arXiv:2103.12607*, Mar. 2021. https://doi.org/10.48550/arXiv.2103.12607.

[60] J. Cai, B. Li, J. Zhang, X. Sun, and B. Chen, "Combine sliced joint graph with graph neural networks for smart contract vulnerability detection," *Journal of Systems and Software*, vol. 195, p. 111550, Jan. 2023. https://doi.org/10.1016/j.jss.2022.111550.

[61] S. M. Beillahi, E. Keilty, K. Nelaturu, A. Veneris, and F. Long, "Automated Auditing of Price Gouging TOD Vulnerabilities in Smart Contracts," in *2022 IEEE*

International Conference on Blockchain and Cryptocurrency (ICBC), pp. 02–05, IEEE. https://doi.org/10.1109/ICBC54727.2022.9805509.

[62] J. Sui, L. Chu, and H. Bao, "An Opcode-Based Vulnerability Detection of Smart Contracts," *Appl. Sci.*, vol. 13, no. 13, 2024. https://doi.org/10.3390/app13137721.

[63] L. S. H. Colin, P. M. Mohan, J. Pan, and P. L. K. Keong, "An Integrated Smart Contract Vulnerability Detection Tool Using Multi-Layer Perceptron on Real-Time Solidity Smart Contracts," *IEEE Access*, vol. 12, pp. 23549–23567, Feb. 2024. https://doi.org/10.1109/ACCESS.2024.3364351.

[64] Y. Yao, H. Li, X. Yang, and Y. Le, "An Improved Vulnerability Detection System of Smart Contracts Based on Symbolic Execution," in *2022 IEEE International Conference on Big Data (Big Data)*, pp. 17–20, IEEE. https://doi.org/10.1109/BigData55660.2022.10020730.

[65] M. Ashouri, "An Extensive Security Analysis on Ethereum Smart Contracts," in *Security and Privacy in Communication Networks*, pp. 144–163, Cham, Switzerland: Springer, Nov. 2021. https://doi.org/10.1007/978-3-030-90019-9_8.

[66] Y. Xu, G. Hu, L. You, and C. Cao, "A Novel Machine Learning-Based Analysis Model for Smart Contract Vulnerability," *Secur. Commun. Netw.*, vol. 2021, p. 5798033, Jan. 2021. https://doi.org/10.1155/2021/5798033.

[67] L. Duan, L. Yang, C. Liu, W. Ni, and W. Wang, "A New Smart Contract Anomaly Detection Method by Fusing Opcode and Source Code Features for Blockchain Services," *IEEE Trans. Network Serv. Manage.*, vol. 20, no. 4, pp. 4354–4368, 2024. https://doi.org/10.1109/TNSM.2023.3278311.

[68] J. Cai, B. Li, T. Zhang, J. Zhang, and X. Sun, "Fine-grained smart contract vulnerability detection by heterogeneous code feature learning and automated dataset construction," *Journal of Systems and Software*, vol. 209, p. 111919, Mar. 2024. https://doi.org/10.1016/j.jss.2023.111919.

[69] M. Dai, Z. Yang, and J. Guo, "SuperDetector: A Framework for Performance Detection on Vulnerabilities of Smart Contracts," *J. Phys. Conf. Ser.*, vol. 2289, p. 012010, June 2022. https://doi.org/10.1088/1742-6596/2289/1/012010.

[70] H. Yang, X. Gu, X. Chen, L. Zheng, and Z. Cui, "CrossFuzz: Cross-contract fuzzing for smart contract vulnerability detection," *Sci. Comput. Programming*, vol. 234, p. 103076, May 2024. https://doi.org/10.1016/j.scico.2023.103076.

[71] J.-W. Liao, T.-T. Tsai, C.-K. He, and C.-W. Tien, "SoliAudit: Smart Contract Vulnerability Assessment Based on Machine Learning and Fuzz Testing," in *2019 Sixth International Conference on Internet of Things: Systems, Management and*

Security (IOTSMS), pp. 22–25, IEEE. https://doi.org/10.1109/IOTSMS48152.2019.8939256.

[72] W. Wang, W. Huang, Z. Meng, Y. Xiong, F. Miao, X. Fang, C. Tu, and R. Ji, "Automated Inference on Financial Security of Ethereum Smart Contracts," in *32nd USENIX Security Symposium (USENIX Security 23)*, pp. 3367–3383, 2023. [https://www.usenix.org/conference/usenixsecurity23/presentation/wang-wansen; accessed 10. Jul. 2024].

[73] S. Zhang, C. Hu, T. Lan, L. Wang, S. Xu, and W. Shao, "Intelligent Contract Vulnerability Detection Method Based on Bic-RL," in *2023 International Conference on Data Security and Privacy Protection (DSPP)*, pp. 128–135, Oct. 2023. https://doi.org/10.1109/DSPP58763.2023.10404628.

[74] S. Hu, T. Huang, F. İlhan, S. F. Tekin, and L. Liu, "Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives," in *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pp. 297–306, Nov. 2023. https://doi.org/10.1109/TPS-ISA58951.2023.00044.

[75] W. Duo, H. Xin, and M. Xiaofeng, "Formal Analysis of Smart Contract Based on Colored Petri Nets," *IEEE Intelligent Systems*, vol. 35, pp. 19–30, May 2020. https://doi.org/10.1109/MIS.2020.2977594.

[76] T. Hu, B. Li, Z. Pan, and C. Qian, "Detect Defects of Solidity Smart Contract Based on the Knowledge Graph," *IEEE Transactions on Reliability*, vol. 73, pp. 186–202, Mar. 2024. https://doi.org/10.1109/TR.2023.3233999.

[77] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, "MANDO-HGT: Heterogeneous Graph Transformers for Smart Contract Vulnerability Detection," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pp. 334–346, May 2023. ISSN: 2574-3864.

[78] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pp. 103–119, Sept. 2021. https://doi.org/10.1109/EuroSP51992.2021.00018.

[79] Z. Wu, S. Li, B. Wang, T. Liu, Y. Zhu, C. Zhu, and M. Hu, "Detecting Vulnerabilities in Ethereum Smart Contracts with Deep Learning," in *2022 4th International Conference on Data Intelligence and Security (ICDIS)*, pp. 55–60, Aug. 2022. https://doi.org/10.1109/ICDIS55630.2022.00016.

[80] H. H. Nguyen, N.-M. Nguyen, C. Xie, Z. Ahmadi, D. Kudendo, T.-N. Doan, and L. Jiang, "MANDO: Multi-Level Heterogeneous Graph Embeddings for Fine-Grained Detection of Smart Contract Vulnerabilities," in *2022 IEEE 9th International Conference on Data Science and Advanced Analytics (DSAA)*, pp. 1–10, Oct. 2022. https://doi.org/10.1109/DSAA54385.2022.10032337.

[81] M. Ren, F. Ma, Z. Yin, H. Li, Y. Fu, T. Chen, and Y. Jiang, "SCStudio: a secure and efficient integrated development environment for smart contracts," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2021, (New York, NY, USA), pp. 666–669, ACM, 2021. https://doi.org/10.1145/3460319.3469078.

[82] G. Crincoli, G. Iadarola, P. E. La Rocca, F. Martinelli, F. Mercaldo, and A. Santone, "Vulnerable Smart Contract Detection by Means of Model Checking," in *Proceedings of the Fourth ACM International Symposium on Blockchain and Secure Critical Infrastructure*, BSCI '22, (New York, NY, USA), pp. 3–10, ACM, 2022. https://doi.org/10.1145/3494106.3528672.

[83] M. Ashouri, "Etherolic: a practical security analyzer for smart contracts," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, SAC '20, (New York, NY, USA), pp. 353–356, Association for Computing Machinery, Mar. 2020. https://dl.acm.org/doi/10.1145/3341105.3374226.

[84] M. Ye, Y. Nan, Z. Zheng, D. Wu, and H. Li, "Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, (New York, NY, USA), pp. 298–309, ACM, 2023. https://doi.org/10.1145/3597926.3598057.

[85] J. Song, H. He, Z. Lv, C. Su, G. Xu, and W. Wang, "An efficient vulnerability detection model for Ethereum smart contracts. International Conference on Network and System Security; 2019: 433-442." https://doi.org/10.1007/978-3-030-36938-5_26.

[86] L. He, X. Zhao, Y. Wang, J. Yang, and X. Sun, "GraphSA: Smart Contract Vulnerability Detection Combining Graph Neural Networks and Static Analysis," in *ECAI 2023*, pp. 1020–1027, IOS Press, 2023. https://doi.org/10.3233/FAIA230374.

[87] L. Zhang, J. Wang, W. Wang, Z. Jin, C. Zhao, Z. Cai, and H. Chen, "A novel smart contract vulnerability detection method based on information graph and ensemble learning," *Sensors*, vol. 22, no. 9, p. 3581, 2022. Publisher: MDPI.

[88] C. Wang, H. Jiang, Y. Wang, Q. Huang, and Z. Zuo, "Research on smart contract vulnerability detection method based on domain features of solidity contracts and attention mechanism," *J. Intell. Fuzzy Syst.*, vol. 45, pp. 1513–1525, Jan. 2023. https://doi.org/10.3233/JIFS-224489.

[89] R. R. Palle, H. Yennapusa, and K. C. R. Kathala, "Enhancing Cloud-Based Smart Contract Security: A Hybrid AI and Optimization Approach for Vulnerability Prediction in FinTech," https://doi.org/10.21275/PR231222115735.

[90] S. Qian, H. Ning, Y. He, and M. Chen, "Multi-Label Vulnerability Detection of Smart Contracts Based on Bi-LSTM and Attention Mechanism," *Electronics*, vol. 11, no. 19, p. 3260, 2022. Publisher: MDPI.

[91] W. Deng, H. Wei, T. Huang, C. Cao, Y. Peng, and X. Hu, "Smart contract vulnerability detection based on deep learning and multimodal decision fusion," *Sensors*, vol. 23, no. 16, p. 7246, 2023. Publisher: MDPI.

[92] Z. Yang and W. Zhu, "Improvement and Optimization of Vulnerability Detection Methods for Ethernet Smart Contracts," *IEEE Access vol. 11*, pp. 78207–78223, 2023. Publisher: IEEE.

[93] X. Sun, L. Tu, J. Zhang, J. Cai, B. Li, and Y. Wang, "*ASSBert*: Active and semi-supervised bert for smart contract vulnerability detection," *Journal of Information Security and Applications*, vol. 73, p. 103423, Mar. 2023. https://doi.org/10.1016/j.jisa.2023.103423.

[94] L. Zhang, J. Wang, W. Wang, Z. Jin, Y. Su, and H. Chen, "Smart contract vulnerability detection combined with multi-objective detection," *Computer Networks*, vol. 217, p. 109289, Nov. 2022. https://doi.org/10.1016/j.comnet.2022.109289.

[95] Z. Zhen, X. Zhao, J. Zhang, Y. Wang, and H. Chen, "DA-GNN: A smart contract vulnerability detection method based on Dual Attention Graph Neural Network," *Computer Networks*, vol. 242, p. 110238, Apr. 2024. https://doi.org/10.1016/j.comnet.2024.110238.

[96] G. Xu, L. Liu, and J. Dong, "Vulnerability Detection of Ethereum Smart Contract Based on SolBERT-BiGRU-Attention Hybrid Neural Model," *CMES - Computer Modeling in Engineering and Sciences*, vol. 137, no. 1, pp. 903–922, 2023. https://doi.org/10.32604/cmes.2023.026627.

[97] J. Zhang, L. Tu, J. Cai, X. Sun, B. Li, W. Chen, and Y. Wang, "Vulnerability detection for smart contract via backward bayesian active learning," in *International Conference on Applied Cryptography and Network Security*, pp. 66–83, Springer, 2022. https://doi.org/10.1007/978-3-031-16815-4_5.

[98] R. Agarwal, T. Thapliyal, and S. K. Shukla, "Vulnerability and transaction behavior based detection of malicious smart contracts," in *Cyberspace Safety and Security: 13th International Symposium, CSS 2021, Virtual Event, November 9–11, 2021, Proceedings 13*, pp. 79–96, Springer, 2022. https://doi.org/10.1007/978-3-030-94029-4_6.

[99] A. Hamdi, L. Fourati, and S. Ayed, "Vulnerabilities and attacks assessments in blockchain 1.0, 2.0 and 3.0: tools, analysis and countermeasures," 2023. https://doi.org/10.1007/s10207-023-00765-0.

[100] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The eye of horus: Spotting and analyzing attacks on Ethereum smart contracts," in *International Conference on Financial Cryptography and Data Security*, pp. 33–52, Springer, 2021. https://doi.org/10.1007/978-3-662-64322-8_2.

[101] Y. Yuan and T. Xie, "Svchecker: A deep learning-based system for smart contract vulnerability detection," in *International Conference on Computer Application and Information Security (ICCAIS 2021)*, vol. 12260, pp. 226–231, SPIE, 2022. https://doi.org/10.1117/12.2637775.

[102] A. Ali, Z. Abideen, and K. Ullah, "SESCon: Secure Ethereum Smart Contracts by Vulnerable Patterns' Detection," *Security and Communication Networks*, vol. 2021, 2021. https://doi.org/10.1155/2021/2897565.

[103] Y. Li, H. Liu, Z. Yang, B. Wang, Q. Ren, L. Wang, and B. Chen, "Protect your smart contract against unfair payment," in *2020 International Symposium on Reliable Distributed Systems (SRDS)*, pp. 61–70, IEEE, 2020. https://doi.org/10.1109/SRDS51746.2020.00014.

[104] J. Sun, P. Yu, and B. Zhang, "Mutation Fuzzy Detection of TOD Vulnerability in Smart Contract," in *Lecture Notes on Data Engineering and Communications Technologies*, vol. 88, pp. 1687–1694, 2021. https://doi.org/10.1007/978-3-030-70665-4_183.

[105] W. Zhang, S. Banescu, L. Pasos, S. Stewart, and V. Ganesh, "Mpro: Combining static and symbolic analysis for scalable testing of smart contract," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 456–462, IEEE, 2019. https://doi.org/10.1109/ISSRE.2019.00052.

[106] Y. Shen, K. Li, L. Mao, W. Li, and X. Li, "Intellicon: Confidence-based approach for fine-grained vulnerability analysis in smart contracts," in *International Conference on Blockchain and Trustworthy Systems*, pp. 45–59, Springer, 2023. https://doi.org/10.1007/978-981-99-8101-4_4.

[107] S. Holler, S. Biewer, and C. Schneidewind, "HoRStify: Sound Security Analysis of Smart Contracts," in *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*, vol. 2023-July, pp. 245–260, 2023. https://doi.org/10.1109/CSF57540.2023.00023.

[108] N. Ashizawa, N. Yanai, J. Cruz, and S. Okamura, "Eth2Vec: Learning Contract-Wide Code Representations for Vulnerability Detection on Ethereum Smart Contracts," in *Proceedings of the 3rd ACM international symposium on blockchain and secure critical infrastructure*, pp. 47–59, 2021. https://doi.org/10.1145/3457337.3457841.

[109] L. J J, K. Singh, and B. Chakravarthi, "Digital forensic framework for smart contract vulnerabilities using ensemble models," *Multimedia Tools and Applications*,

vol. 83, no. 17, pp. 51469–51512, 2023. https://doi.org/10.1007/s11042-023-17308-3.

[110] V. Jain and M. Tripathi, "An integrated deep learning model for Ethereum smart contract vulnerability detection," *International Journal of Information Security*, vol. 23, no. 1, pp. 557–575, 2024. https://doi.org/10.1007/s10207-023-00752-5.

[111] J. Song, H. He, Z. Lv, C. Su, G. Xu, and W. Wang, "An Efficient Vulnerability Detection Model for Ethereum Smart Contracts," in *Network and System Security: 13th International Conference, NSS 2019, Sapporo, Japan, December 15–18, 2019, Proceedings 13*, vol. 11928 LNCS, pp. 433–442, 2019. https://doi.org/10.1007/978-3-030-36938-5_26.

[112] Z. Xu, X. Chen, X. Dong, H. Han, Z. Yan, K. Ye, C. Li, Z. Zheng, H. Wang, and J. Zhang, "An Efficient Code-Embedding-Based Vulnerability Detection Model for Ethereum Smart Contracts," *International Journal of Data Warehousing and Mining*, vol. 19, no. 2, pp. 1–23, 2023. https://doi.org/10.4018/IJDWM.320473.

[113] D. Han, Q. Li, L. Zhang, and T. Xu, "A smart contract vulnerability detection model based on graph neural networks," in *2022 4th International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, pp. 834–837, 2022. https://doi.org/10.1109/ICFTIC57696.2022.10075325.

[114] H. H. Nguyen, N.-M. Nguyen, H.-P. Doan, Z. Ahmadi, T.-N. Doan, and L. Jiang, "MANDO-GURU: vulnerability detection for smart contract source code by heterogeneous graph embeddings," in *ACM Conferences*, pp. 1736–1740, New York, NY, USA: ACM, Nov. 2022. https://doi.org/10.1145/3540250.3558927.

[115] L. Heimbach and R. Wattenhofer, "SoK: Preventing Transaction Reordering Manipulations in Decentralized Finance," in *Proceedings of the 4th ACM Conference on Advances in Financial Technologies*, pp. 47–60, Mar. 2022. https://doi.org/10.1145/3558535.3559784.

[116] M. Barczentewicz, "MEV on Ethereum: A Policy Analysis," Jan. 2023. [https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4332703; accessed 3. Aug. 2024].

[117] S. Yang, F. Zhang, K. Huang, X. Chen, Y. Yang, and F. Zhu, "SoK: MEV Countermeasures: Theory and Practice," *arXiv*, Dec. 2022. https://doi.org/10.48550/arXiv.2212.05111.

[118] "Ethereum is a Dark Forest," Aug. 2020. [https://www.paradigm.xyz/2020/08/ethereum-is-a-dark-forest; accessed 12. Jul. 2024].

[119] B. Weintraub, C. F. Torres, C. Nita-Rotaru, and R. State, "A Flash(bot) in the Pan: Measuring Maximal Extractable Value in Private Pools," in *Proceedings of the 22nd ACM Internet Measurement Conference*, pp. 458–471, Oct. 2022. [Online; accessed 3. Aug. 2024].

[120] "Transaction Flow Chart | EigenTx:0x315799864c378f0d02a66bcdc5308bc92430d50e92a9f9a8a0e38694e3565 Aug. 2024. [https://www.eigenphi.io/mev/eigentx/0x315799864c378f0d02a66bcdc5308bc92430d50e92a9f9a8a0e38694e3565819,0x168df43a6ac8a144d538147bfc26a7f27275e6ca9ecc705b979a629a4d804057,0x265637e9ec433d8a9022dec9c5e91f6b668fecedc18521b513eadd0dd9f414fa?mevId=0xedf0b223a3f98164cc6643e5596d340614cb3fa18ec42fcaac403ac503b40f39; accessed 3. Aug. 2024].

[121] L. Heimbach, E. Schertenleib, and R. Wattenhofer, "DeFi Lending During The Merge," *arXiv*, Jan. 2023. https://doi.org/10.48550/arXiv.2303.08748.

[122] "Sandwich Overview | EigenPhi," Dec. 2023. [https://www.eigenphi.io/mev/ethereum/sandwich accessed 10. Dec. 2023].

[123] Z. Alipanahloo, A. S. Hafid, and K. Zhang, "Maximal extractable value mitigation approaches in ethereum and layer-2 chains: A comprehensive survey," *arXiv preprint arXiv:2407.19572*, 2024. [https://doi.org/10.48550/arXiv.2407.19572.

[124] "Proposer-builder separation | ethereum.org," Aug. 2024. [https://ethereum.org/en/roadmap/pbs; accessed 3. Aug. 2024].

[125] "fastlane-bot," Aug. 2024. [https://github.com/bancorprotocol/fastlane-bot; accessed 10. Aug. 2024].

[126] "libMEV," Aug. 2024. [https://libmev.com; accessed 9. Aug. 2024].

[127] etherscan. io, "jaredfromsubway: MEV Bot | Address 0x6b75d8af000000e20b7a7ddf000ba900b4009a80 | Etherscan," Aug. 2024. [https://etherscan.io/address/0x6b75d8af000000e20b7a7ddf000ba900b4009a80; accessed 9. Aug. 2024].

[128] "libMEV," Aug. 2024. [https://libmev.com/bundles?presetRange=%22SINCE+MERGE%22&mevTypeLabels=%5B%22Arbitrage%22%2C%22Sandwich%22%2C%22Liquidation%22%5D&tokenFilters=%5B%5D&contractFilters=%5B%220x6b75d8af000000e20b7a7ddf000ba900b4009a80%22%5D&orderBy=%22profit_usdc%22&orderDirection=%22desc%22; accessed 9. Aug. 2024].

[129] T. Copeland, "Jaredfromsubway.eth's MEV bot rakes in millions of dollars in three months," *Block*, May 2023. [https://www.theblock.co/post/230218/jaredfromsubway-mev-bot; accessed 9. Aug. 2024].

[130] L. Heimbach, L. Kiffer, C. Ferreira Torres, and R. Wattenhofer, "Ethereum's Proposer-Builder Separation: Promises and Realities," in *ACM Conferences*, pp. 406–420, New York, NY, USA: ACM, Oct. 2023. https://doi.org/10.1145/3618257.3624824.

[131] D. Grandjean, L. Heimbach, and R. Wattenhofer, "Ethereum Proof-of-Stake Consensus Layer: Participation and Decentralization," *arXiv*, June 2023. https://doi.org/10.48550/arXiv.2306.10777.

[132] "Lido Liquid Staking," Aug. 2024. [https://lido.fi; accessed 9. Aug. 2024].

[133] "Relay Landscape | Ethereum Mainnet," Aug. 2024. [https://explorer.rated.network/relays?network=mainnet&timeWindow=all; accessed 6. Aug. 2024].

[134] "Two-slot proposer/builder separation - Proof-of-Stake - Ethereum Research," Oct. 2021. [https://ethresear.ch/t/two-slot-proposer-builder-separation/10980; accessed 9. Aug. 2024].

[135] "Builder Landscape | Ethereum Mainnet," Aug. 2024. [https://explorer.rated.network/builders?network=mainnet&timeWindow=30d&page=1; accessed 9. Aug. 2024].

[136] M. Kelkar, F. Zhang, S. Goldfeder, and A. Juels, "Order-fairness for byzantine consensus," in *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part III 40*, pp. 451–480, Springer, 2020. https://doi.org/10.1007/978-3-030-56877-1_16.

[137] A. Mamageishvili, M. Kelkar, J. C. Schlegel, and E. W. Felten, "Buying Time: Latency Racing vs. Bidding in Transaction Ordering," *arXiv*, June 2023. https://doi.org/10.48550/arXiv.2306.02179.

[138] "Sequencer node | Metis Developer Documentation," Aug. 2024. [https://docs.metis.io/dev/protocol/overview/sequencer-node; accessed 5. Aug. 2024].

[139] "What is a multi-party computation (MPC) wallet?," Aug. 2024. [https://www.alchemy.com/overviews/mpc-wallet; accessed 5. Aug. 2024].

[140] M. Kelkar, S. Deb, S. Long, A. Juels, and S. Kannan, "Themis: Fast, Strong Order-Fairness in Byzantine Consensus," in *ACM Conferences*, pp. 475–489, New York, NY, USA: ACM, Nov. 2023. https://doi.org/10.1145/3576915.3616658.

[141] "Shutter | Protecting You From Sandwich Attacks and Voting Whales," Aug. 2024. [https://www.shutter.network; accessed 10. Aug. 2024].

[142] H. Zhang, L.-H. Merino, V. Estrada-Galiñanes, and B. Ford, "Flash Freezing Flash Boys: Countering Blockchain Front-Running," in *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*, p. 10, IEEE. https://doi.org/10.1109/ICDCSW56584.2022.00026.

[143] "Radius," Aug. 2024. [https://www.theradius.xyz; accessed 10. Aug. 2024].

[144] I. Bentov, Y. Ji, F. Zhang, L. Breidenbach, P. Daian, and A. Juels, "Tesseract: Real-Time Cryptocurrency Exchange Using Trusted Hardware," in *ACM Conferences*, pp. 1521–1538, New York, NY, USA: ACM, Nov. 2019. https://doi.org/10.1145/3319535.3363221.

[145] L. Breidenbach, P. Daian, F. Tramèr, and A. Juels, "Enter the Hydra: Towards Principled Bug Bounties and Exploit-Resistant Smart Contracts," pp. 1335–1352, 2018. [https://www.usenix.org/conference/usenixsecurity18/presentation/breindenbach; accessed 10. Aug. 2024].

[146] "CoW Protocol | CoW Protocol Documentation," Aug. 2024. [https://docs.cow.fi/cow-protocol; accessed 5. Aug. 2024].

[147] M. Ciampi, M. Ishaq, M. Magdon-Ismail, R. Ostrovsky, and V. Zikas, "FairMM: A Fast and Frontrunning-Resistant Crypto Market-Maker," in *International Symposium on Cyber Security, Cryptology, and Machine Learning*, pp. 428–446, Springer, 2022. https://doi.org/10.1007/978-3-031-07689-3_31.

[148] "Hashflow 2.0: Best Prices. Deep Liquidity. Every Token.," Aug. 2024. [https://www.hashflow.com; accessed 10. Aug. 2024].

[149] "Welcome to Flashbots | Flashbots Docs," June 2024. [https://docs.flashbots.net; accessed 6. Aug. 2024].

[150] "mev-boost," Aug. 2024. [https://github.com/flashbots/mev-boost; accessed 6. Aug. 2024].

[151] "Overview | Flashbots Docs," Aug. 2024. [https://docs.flashbots.net/flashbots-mev-boost/introduction; accessed 3. Aug. 2024].

[152] "MEV-Share: programmably private orderflow to share MEV with users - The Flashbots Ship - The Flashbots Collective," Feb. 2023. [https://collective.flashbots.net/t/mev-share-programmably-private-orderflow-to-share-mev-with-users/1264; accessed 10. Aug. 2024].

[153] P. Lanningham and SSTeam, "Sundaeswap fundamentals," June 2021. [https://ouroboros.mobi/wp-content/uploads/2021/10/SundaeSwap-2021-06-01-Fundamentals.pdf; accessed 19. Aug. 2024].

[154] United States, "Indictment of anton peraire-bueno, and james peraire-bueno." Case No. 24-CRIM-293, 2024. https://www.justice.gov/opa/media/1351996/dl, accessed 6. Aug. 2024.

[155] S. Kessler and N. De, "Brothers Accused of $25M Ethereum Exploit as U.S. Reveals Fraud Charges," *CoinDesk*, May 2024.

[156] R. Behnke, "Explained: The Ronin Network Hack (August 2024)," *https://www.facebook.com/Halbornsec/*, Aug. 2024. [https://www.halborn.com/blog/post/explained-the-ronin-network-hack-august-2024; accessed 9. Aug. 2024].

[157] etherscan. io, "Ethereum Transaction Hash (Txhash) Details | Etherscan," Aug. 2024. [https://etherscan.io/tx/0x2619570088683e6cc3a38d93c3d98899e5783864e15525d5f5810c11189ba6cb; accessed 9. Aug. 2024].

[158] R. Behnke, "Explained: The Vyper Bug Hack (July 2023)," *https://www.facebook.com/Halbornsec/*, Aug. 2023. [https://www.halborn.com/blog/post/explained-the-vyper-bug-hack-july-2023; accessed 9. Aug. 2024].

[159] S. Sriram, "Curve Exploit Results in Largest MEV Block Rewards in Ethereum's History - Unchained," *Unchained*, July 2023. [https://unchainedcrypto.com/curve-exploit-results-in-largest-mev-block-rewards-in-ethereums-history; accessed 9. Aug. 2024].