Noninterference Analysis for Smart Contracts: Would you Bet on it?

Samia Guesmi[1,*], Carla Piazza[2] and Sabina Rossi[3]

[1]*Università degli Studi di Camerino*
[2]*Università degli Studi di Udine*
[3]*Università Ca' Foscari di Venezia*

Abstract

In the context of Blockchains and Decentralized Finance the notion of Maximal Extractable Value (MEV) is attracting more and more attention. MEV is the maximum gain that users–including miners and validators—can obtain by interacting with a smart contract and with its dependencies. Such profits witness attacks that also exploit strategic transaction manipulations (e.g., reordering transactions in blocks) and distort the meaning of smart contracts. The use of the notion of noninterference for modeling and analysing MEV attacks has recently been proposed in the literature. Noninterference aims to capture unwanted information flows in multi-level systems. Various definitions of noninterference have been presented, and among these those based on unwinding conditions allow the possible flows to be specifically located in a system. In this paper we investigate the use of such unwinding conditions to analyze MEV. We exploit a simple case study–the Bet contract–to highlight the advantages and disadvantages of our proposal.

Keywords

Smart Contracts, MEV, Noninterference, Unwinding Conditions

Introduction

Decentralized Finance (DeFi) is revolutionizing the landscape of the global digital economy by amalgamating decentralization and distribution principles with traditional financial services. This integration operates within a decentralized framework, leveraging digital assets primarily through blockchain technology [1, 2, 3]. Such pioneering methods engender an alternative financial system distinguished by its emphasis on decentralization, fostering innovation, promoting interoperability, and ensuring transparency [4]. Ethereum, renowned for its sophisticated smart contract functionality, stands as the cornerstone technology supporting various DeFi services. Smart contracts serve as automated enforcers of contractual terms, allowing for intricate interactions between parties without the need for intermediaries [5].

Despite the security assurances provided by blockchain technology, DeFi smart contracts possess inherent vulnerabilities as highlighted in recent studies [6]. Within the DeFi ecosystem, decentralized exchanges (DEXs) play a central role by facilitating trustless transactions through smart contracts, eliminating the need for centralized intermediaries. Automated Market Maker (AMM) protocols, a prominent feature of DEXs, have garnered significant attention in the wake

6th Distributed Ledger Technologies Workshop (DTL2024)
*Corresponding author.
semia.guesmi@unicam.it (S. Guesmi)

© 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

of increased interest in blockchain technology. These protocols operate on a peer-to-pool basis, wherein liquidity providers contribute assets to liquidity pools, enabling users to exchange assets at algorithmically determined prices [4].

In the realm of Blockchains and Decentralized Finance (DeFi), the concept of Maximal Extractable Value (MEV) is gaining increasing attention. MEV represents the maximum potential gain that users, including miners and validators, can achieve through interactions with a smart contract and its associated dependencies [7]. This potential for profit attracts not only legitimate users but also malicious actors who exploit strategic transaction manipulations, such as reordering transactions within blocks, to manipulate the outcomes of smart contracts. These actors often employ automated bots to engage in frontrunning, exploiting transactional intricacies for potential gains [8]. Notable examples of MEV exploitation include instances of decentralized exchange (DEX) arbitrage, where participants exploit price differences across decentralized exchange protocols [9]. Such attacks not only compromise the integrity of smart contracts but also distort their intended functionality.

Recent literature has proposed utilizing the concept of noninterference for modeling and analyzing MEV attacks. Noninterference aims to identify and mitigate undesired information flows within multi-level systems. Various definitions of noninterference have been introduced [10, 11, 12, 13], with those based on unwinding conditions particularly useful for pinpointing potential information flows within a system [14, 15, 16]. Our study builds upon prior research, particularly drawing from the contributions by Babel et al. in [17] and by Bartoletti et al. in [18]. Specifically, Bartoletti et al.'s work introduces a novel concept of secure composability for smart contracts, aiming to safeguard compound contracts from economic harm caused by adversaries interfering with their dependencies.

In this paper, our investigation centers on the application of unwinding conditions to analyze the Maximal Extractable Value (MEV). Our approach involves a meticulous examination of a straightforward case study—the Bet contract—with the aim of elucidating both the strengths and limitations of our methodology. Specifically, we utilize a simple imperative concurrent language, as introduced in a previous study [14, 16] for security verification, to identify vulnerabilities within DeFi smart contracts that are susceptible to MEV interferences. Through the presentation of the Bet contract case study, we spotlight scenarios where particular conditions affect MEV noninterference, thereby emphasizing potential sources of vulnerability within DeFi services. Differently from the proposal of [18], we exploit the formalization of noninterference in terms of unwinding conditions. This allows us to abstract from the environment, namely the specific implementations of smart contracts that interact with the one under examination. This abstraction enables a more comprehensive evaluation of MEV vulnerabilities within DeFi systems.

Structure of the paper. The structure of the paper is as follows: In Section 1, we present the case study of the Bet contract. Section 2 introduces the concept of noninterference for an imperative concurrent language and outlines the use of downgrading to capture the release of sensitive information. Section 3 details the modeling of the Bet contract within our imperative language, including the capture of the noninterference property and the application of downgrading functionality to differentiate secure scenarios from potentially risky ones. Finally, Section 4 discusses related work and concludes the paper.

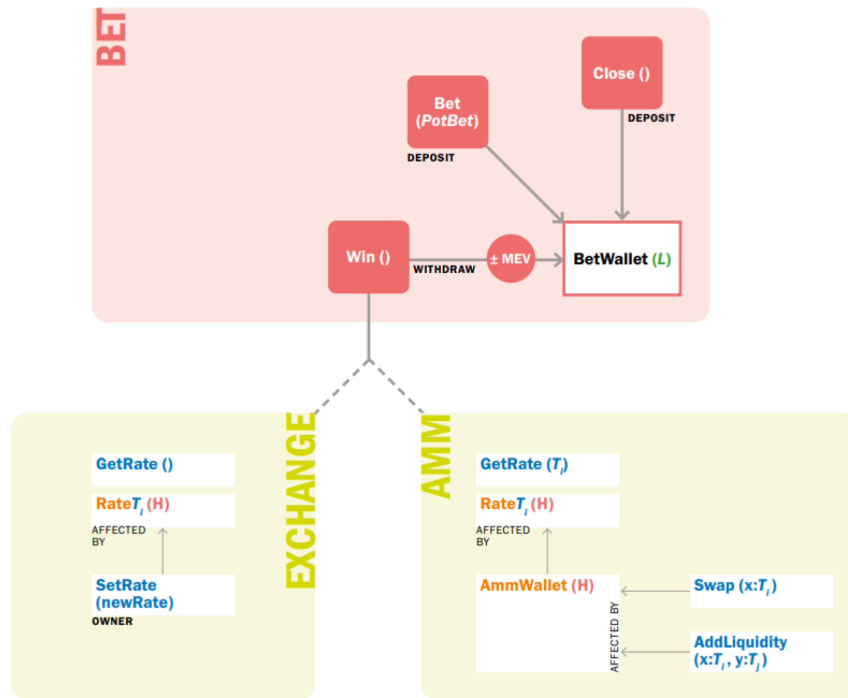


Figure 1: MEV Insight: Navigating Bet Connections

1. The Bet Contract Case Study

In this section we briefly introduce the main functionalities of a betting service, as also detailed in [18], which serves as a straightforward case study in this paper to demonstrate how to instantiate noninterference expressed through unwinding conditions for detecting MEV attacks. Specifically, the Bet contract we consider enables individuals to place bets on the exchange rate between a given token and Ether. This contract is parameterized based on an oracle contract responsible for providing token price information. Upon deployment, the owner initiates the bet contract with an initial pot of Ether; for joining, a player is required to contribute an amount of Ether equal to the pot. Prior to the deadline, the player can withdraw the pot if the oracle exchange rate is greater than the bet rate defined by the Bet contract owner. Subsequently, after the deadline elapses, the pot is transferred to the owner's wallet.

We explore two instances of the oracle contract, with the first being an Automated Market Maker (AMM) inspired by the Uniswap protocol (see, `https://uniswap.org/`). This AMM allows users to both add liquidity for two tokens and execute swaps between the held tokens. Additionally, the contract enables the querying of token pairs and exchange rates. In contrast, the second oracle contract, referred to as the Exchange contract, differs from the AMM contract in terms of rate setting. In the AMM, the rate is affected by the fluctuations in the total amount of tokens staked in the AMM wallet, equating to the division of the total amount of one token by the total amount of the paired token. Conversely, in the Exchange contract, the rate is solely

determined by the owner and remains unaffected by variations in wallet values resulting from user interactions. Figure 1 illustrates the primary functionalities of the Bet contract and its potential interactions with either AMM or Exchange.

As detailed in [18], MEV attacks become feasible when the Bet contract employs AMM as its oracle. In this scenario, an attacker can place a bet and subsequently interact with the AMM contract before executing the function to claim the win. By doing so, the attacker alters the exchange rate, thereby influencing the outcome of the bet. This vulnerability is not limited to regular users; its success rate significantly increases when the attacker is a miner with the ability to dictate transaction order within blocks. Conversely, if the Bet contract relies on the Exchange contract as its oracle, the attacker cannot modify the rate, even if interaction with the Exchange contract occurs.

In the forthcoming sections, we will show that when we use noninterference for modeling this attack the wallet of the bet contract is going to play the role of the low level variable over which we observe a flow of information (denoted by a green L in Figure 1). Such flow of information comes from the high level variable representing the exchange rate (denoted by a red H in Figure 1), i.e., from a variable whose value can be altered by other contracts. Our approach offers the advantage of focusing solely on the bet contract and identifying potentially dangerous instructions within it. Then, a thorough analysis of the oracles becomes imperative to distinguish between the two scenarios.

2. Noninterference and Downgrading over a Concurrent Imperative Language

In [16] we considered an imperative concurrent language, inspired by the one introduced in [19] and proposed different notions of noninterference with the aim of ensuring that in multi-level systems, where users are categorized as high (e.g., system administrators) and low (e.g., standard users) no information flow occurs from the high level to the low one. A flow of information from high to low could in fact represent the public disclosure of private data.

Recognizing that completely preventing any potential flow might be overly restrictive in many scenarios, we also proposed a downgrading mechanism in [16]. This mechanism enables explicit allowance of delimited flows, providing a nuanced approach to managing information flow within the system.

In this section, we provide a brief overview of those papers. However, to maintain simplicity in our presentation, we refrain from utilizing specific observational equivalences, like, e.g., bisimulation. Instead, we solely focus on observing the values of low-level variables at the end of the execution.

2.1. The Language

Let $\mathbb{Z}$ be the set of integer numbers, $\mathbb{T} = \{\text{true}, \text{false}\}$ be the set of boolean values, $\mathbb{L}$ be a set of low level locations and $\mathbb{H}$ be a set of high level locations, with $\mathbb{L} \cap \mathbb{H} = \emptyset$. The set **Aexp** of arithmetic expressions is defined by the grammar:

$$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$$

where $n \in \mathbb{Z}$ and $X \in \mathbb{L} \cup \mathbb{H}$. We assume that arithmetic expressions are total. The set **Bexp** of boolean expressions is defined by:

$$b ::= \text{true} \mid \text{false} \mid (a_0 = a_1) \mid (a_0 \leq a_1) \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1$$

where $a_0, a_1 \in \mathbf{Aexp}$.

We say that an arithmetic expression a is *confidential*, denoted by $a \in \texttt{high}$, if there is a high level location which occurs in it. Otherwise we say that a is *public*, denoted by $a \in \texttt{low}$. Similarly, we say that a boolean expression b is *confidential*, denoted by $b \in \texttt{high}$, if there is a confidential arithmetic expression which occurs in it. Otherwise we say that b is *public*, denoted by $b \in \texttt{low}$. This notion of confidentiality, both for arithmetic and boolean expressions, is purely syntactic. Notice that a high level expression can contain low level locations, i.e., its value can depend on the values of low level locations. This reflects the idea that a high level user can read both high and low level data.

The set **Prog** of programs of our language is defined as:

$$S \ ::= \ \texttt{skip} \mid X := a \mid S_0; S_1$$
$$P \ ::= \ S \mid P_0; P_1 \mid \texttt{if}(b)\,\{P_0\}\,\texttt{else}\,\{P_1\} \mid \texttt{while}(b)\,\{P\} \mid \texttt{await}(b)\,\{S\} \mid \texttt{co}\ P_1 \| \ldots \| P_n\ \texttt{oc}$$

where $a \in \mathbf{Aexp}$, $X \in \mathbb{L} \cup \mathbb{H}$, and $b \in \mathbf{Bexp}$. Notice that, as in [19], in the body of the `await` operator only sequences of assignments are allowed.

The operational semantics of our language is based on the notion of *state*. A state σ is a function which assigns to each location an integer, i.e., $\sigma : \mathbb{L} \cup \mathbb{H} \longrightarrow \mathbb{Z}$. Given a state σ, we denote by $\sigma[X/n]$ the state σ' such that $\sigma'(X) = n$ and $\sigma'(Y) = \sigma(Y)$ for all $Y \neq X$. Moreover, we denote by σ_L the restriction of σ to the low level locations and we write $\sigma =_l \theta$ for $\sigma_L = \theta_L$.

Given an arithmetic expression $a \in \mathbf{Aexp}$ and a state σ, the evaluation of a in σ, denoted by $\langle a, \sigma \rangle \rightarrow n$ with $n \in \mathbb{Z}$, is defined in the standard way. Similarly, $\langle b, \sigma \rangle \rightarrow v$ with $b \in \mathbf{Bexp}$ and $v \in \{\text{true}, \text{false}\}$, denotes the evaluation of a boolean expression b in a state σ. In both cases atomicity of the evaluation operation is assumed.

Our operational semantics is expressed in terms of state transitions. A transition from a program P and a state σ has the form $\langle P, \sigma \rangle \stackrel{\epsilon}{\rightarrow} \langle P', \sigma' \rangle$ where P' is either a program or the special symbol end (denoting termination) and $\epsilon \in \{\texttt{high}, \texttt{low}\}$ stating that the transition is either confidential or public. The operation $\epsilon_1 \cup \epsilon_2$ returns `low` if both ϵ_1 and ϵ_2 are `low` otherwise it returns `high`. Let $\mathbb{P} = \mathbf{Prog} \cup \{\text{end}\}$ and Σ be the set of all the possible states. The operational semantics of $\langle P, \sigma \rangle \in \mathbb{P} \times \Sigma$ is the *labelled transition system* (LTS) defined by structural induction on P according to the rules depicted in Table 1. Intuitively, the semantics of the sequential composition imposes that a program of the form $P_0; P_1$ behaves like P_0 until P_0 terminates and then it behaves like P_1. To describe the semantics of a program of the form $\texttt{while}(b)\,\{P\}$ we have to distinguish two cases: if b is true, then the program is unravelled to $P; \texttt{while}(b)\,\{P\}$; otherwise it terminates. As far as the `await` operator is concerned, if b is true then $\texttt{await}(b)\,\{P\}$ terminates executing P in one indivisible action (as an atomic block), i.e., it is not possible to observe the state changes internal to the execution of P, while if b is false then $\texttt{await}(b)\,\{P\}$ loops waiting for b to become true. Finally, in a parallel composition of the form $\texttt{co}\ P_0 \| \ldots \| P_n \texttt{oc}$ any of the P_i can move, and the termination is reached only when all the P_i's have terminated.

$$\frac{}{\langle \mathtt{skip}, \sigma \rangle \xrightarrow{\mathtt{low}} \langle \mathrm{end}, \sigma \rangle} \qquad \frac{\langle a, \sigma \rangle \to n}{\langle X := a, \sigma \rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma[X/n] \rangle}\ a \in \epsilon$$

$$\frac{\langle P_0, \sigma \rangle \xrightarrow{\epsilon} \langle P_0', \sigma' \rangle}{\langle P_0; P_1, \sigma \rangle \xrightarrow{\epsilon} \langle P_0'; P_1, \sigma' \rangle}\ P_0' \not\equiv \mathrm{end} \qquad \frac{\langle P_0, \sigma \rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma' \rangle}{\langle P_0; P_1, \sigma \rangle \xrightarrow{\epsilon} \langle P_1, \sigma' \rangle}$$

$$\frac{\langle b, \sigma \rangle \to \mathrm{true}}{\langle \mathtt{if}(b)\,\{P_0\}\ \mathtt{else}\ \{P_1\}, \sigma \rangle \xrightarrow{\epsilon} \langle P_0, \sigma \rangle}\ b \in \epsilon \qquad \frac{\langle b, \sigma \rangle \to \mathrm{false}}{\langle \mathtt{if}(b)\,\{P_0\}\ \mathtt{else}\ \{P_1\}, \sigma \rangle \xrightarrow{\epsilon} \langle P_1, \sigma \rangle}\ b \in \epsilon$$

$$\frac{\langle b, \sigma \rangle \to \mathrm{true}}{\langle \mathtt{while}(b)\,\{P\}, \sigma \rangle \xrightarrow{\epsilon} \langle P; \mathtt{while}(b)\,\{P\}, \sigma \rangle}\ b \in \epsilon \qquad \frac{\langle b, \sigma \rangle \to \mathrm{false}}{\langle \mathtt{while}(b)\,\{P\}, \sigma \rangle \xrightarrow{\epsilon} \langle \mathrm{end}, \sigma \rangle}\ b \in \epsilon$$

$$\frac{\langle b, \sigma \rangle \to \mathrm{true} \quad \langle S, \sigma \rangle \overset{\epsilon_2}{\rightsquigarrow} \langle \mathrm{end}, \sigma' \rangle}{\langle \mathtt{await}(b)\,\{S\}, \sigma \rangle \xrightarrow{\epsilon_1 \cup \epsilon_2} \langle \mathrm{end}, \sigma' \rangle}\ b \in \epsilon_1 \qquad \frac{\langle b, \sigma \rangle \to \mathrm{false}}{\langle \mathtt{await}(b)\,\{S\}, \sigma \rangle \xrightarrow{\epsilon} \langle \mathtt{await}(b)\,\{S\}, \sigma \rangle}\ b \in \epsilon$$

$$\frac{\langle P_i, \sigma \rangle \xrightarrow{\epsilon} \langle P_i', \sigma' \rangle}{\langle \mathrm{co}\ P_1 \| \dots \| P_i \| \dots \| P_n\ \mathrm{oc}, \sigma \rangle \xrightarrow{\epsilon} \langle \mathrm{co}\ P_1 \| \dots \| P_i' \| \dots \| P_n\ \mathrm{oc}, \sigma' \rangle} \qquad \frac{}{\langle \mathrm{co\ end} \| \dots \| \mathrm{end} \| \dots \| \mathrm{end\ oc}, \sigma \rangle \xrightarrow{\mathtt{low}} \langle \mathrm{end}, \sigma \rangle}$$

Table 1
The operational semantics.

We use the following notations. We write $\langle P, \sigma \rangle \to \langle P', \sigma' \rangle$ to denote $\langle P, \sigma \rangle \xrightarrow{\epsilon} \langle P', \sigma' \rangle$ with $\epsilon \in \{\mathtt{low}, \mathtt{high}\}$ and $\langle P_0, \sigma_0 \rangle \to^n \langle P_n, \sigma_n \rangle$ with $n \geq 0$ for $\langle P_0, \sigma_0 \rangle \to \langle P_1, \sigma_1 \rangle \to \cdots \to \langle P_{n-1}, \sigma_{n-1} \rangle \to \langle P_n, \sigma_n \rangle$. The notation $\langle P_0, \sigma_0 \rangle \overset{\mathtt{low}}{\rightsquigarrow} \langle P_n, \sigma_n \rangle$ stands for $\langle P_0, \sigma_0 \rangle \to^n \langle P_n, \sigma_n \rangle$ for some $n \geq 0$ with all the n transitions labelled with $\mathtt{low}$; similarly $\langle P_0, \sigma_0 \rangle \overset{\mathtt{high}}{\rightsquigarrow} \langle P_n, \sigma_n \rangle$ stands for $\langle P_0, \sigma_0 \rangle \to^n \langle P_n, \sigma_n \rangle$ for some $n \geq 0$ with at least one of the n transitions labelled with $\mathtt{high}$. Finally, we write $\langle P, \sigma \rangle \rightsquigarrow \langle P', \sigma' \rangle$ to denote $\langle P, \sigma \rangle \overset{\epsilon}{\rightsquigarrow} \langle P', \sigma' \rangle$ with $\epsilon \in \{\mathtt{low}, \mathtt{high}\}$.

Notice that the operational semantics defined in Table 1 is non-deterministic, since in the case of parallel composition there are many possible evolutions depending on which component is selected.

2.2. Noninterference

Intuitively, when we analyze a program P that involves both low and high-level variables, our objective is to ensure that regardless of how the high-level variables are modified during execution by other high-level components, which represent interactions with high-level users, the values of the low-level variables remain unaffected. Put simply, if actions performed by other components on the high-level variables lead to changes in the low-level ones, it signifies an unwanted information flow, or interference. In essence, we focus solely on the program P and aim to demonstrate its security in any possible context or environment – meaning, irrespective of the actions taken by high-level users. This concept has been formalized in [16], where we

introduced a *generalized unwinding condition* to define classes of programs that are parametric with respect to:

- a binary relation $\doteq$ which equates two states if they are indistinguishable for a low level observer;
- a binary reachability relation $\mathscr{R}$ on $\mathbb{P} \times \Sigma$ which associates to each pair $\langle P, \sigma \rangle$ all the pairs $\langle F, \psi \rangle$ which, in some sense, are reachable from $\langle P, \sigma \rangle$.
- a binary relation $\div$ which equates two pairs $\langle P, \sigma \rangle$ and $\langle Q, \theta \rangle$ if they are indistinguishable for a low level observer;

A pair $\langle P, \sigma \rangle$ satisfies (an instance of) our unwinding framework (i.e., there are no flows of information from high to low) if any high level step $\langle F, \psi \rangle \stackrel{\texttt{high}}{\to} \langle G, \varphi \rangle$ performed by a pair $\langle F, \psi \rangle$ reachable from $\langle P, \sigma \rangle$ has no effect on the observation of a low level user. This is achieved by requiring that all the elements in the set $\{\langle F, \pi \rangle \mid \pi \doteq \psi\}$ (whose states are low level equivalent) may perform a transition reaching an element of the set $\{\langle R, \rho \rangle \mid \langle R, \rho \rangle \div \langle G, \varphi \rangle\}$ (whose elements are all indistinguishable for a low level observer). We use the notation $\mathscr{R}(\langle P, \sigma \rangle)$ to denote the set of pairs reachable from $\langle P, \sigma \rangle$, i.e., $\mathscr{R}(\langle P, \sigma \rangle) = \{\langle F, \psi \rangle \mid \langle P, \sigma \rangle \mathscr{R} \langle F, \psi \rangle\}$.

Definition 1. (Generalized Unwinding) Let $\doteq$ be a binary relation over Σ, $\mathscr{R}$ and $\div$ be two binary relations over $\mathbb{P} \times \Sigma$. We define the *unwinding class* $\mathscr{W}(\doteq, \mathscr{R}, \div)$ by:

$$
\begin{aligned}
\mathscr{W}(\doteq, \mathscr{R}, \div) \stackrel{\text{def}}{=} \{\langle P, \sigma \rangle \in \mathbf{Prog} \times \Sigma \mid\; & \forall \langle F, \psi \rangle \in \mathscr{R}(\langle P, \sigma \rangle) \\
& \text{if } \langle F, \psi \rangle \stackrel{\texttt{high}}{\to} \langle G, \varphi \rangle \text{ then} \\
& \forall \pi \in \Sigma \text{ such that } \pi \doteq \psi,\; \exists \langle R, \rho \rangle : \\
& \langle F, \pi \rangle \to \langle R, \rho \rangle \text{ and } \langle G, \varphi \rangle \div \langle R, \rho \rangle\}
\end{aligned}
$$

We will now apply the concept of generalized unwinding in one of its simplest forms, which will serve as a sufficient foundation for our analysis in the subsequent section focusing on our case study. As far as the relation $\doteq$ is concerned, it is quite natural to consider two states to be equivalent when they assign the same values to the low level variables, i.e., we consider $\doteq$ to be the relation $=_l$. The relation $\mathscr{R}$ is the notion of reachability we rely on, i.e., $\rightsquigarrow$ is defined by the operational semantics. In [16], we introduced the concept of low-level bisimulation to define what the low-level user can observe. Specifically, we utilized low-level bisimulation to instantiate the equivalence relation denoted by $\div$. Bisimulation is the appropriate notion to employ when it is assumed that the low-level user can observe the values of low-level variables at any point during the execution. However, for the sake of simplicity in our current presentation, we make the assumption that the low-level user can only observe the values of variables at the end of the execution. We are aware of the fact that this is not a good choice when non-terminating programs are considered. Formally this means that we instantiate $\div$ as $\approx_l$ defined as follows.

Definition 2 ($\approx_l$)**.** Let $\langle G, \varphi \rangle$ and $\langle R, \rho \rangle$ be two pairs. It holds that $\langle G, \varphi \rangle \approx_l \langle R, \rho \rangle$ if and only if whenever $\langle G, \varphi \rangle \rightsquigarrow \langle \texttt{end}, \varphi' \rangle$, there exists a pair $\langle \texttt{end}, \rho' \rangle$ such that $\langle R, \rho \rangle \rightsquigarrow \langle \texttt{end}, \rho' \rangle$ with $\varphi' =_l \rho'$, and vice-versa (i.e., $\approx_l$ is symmetric).

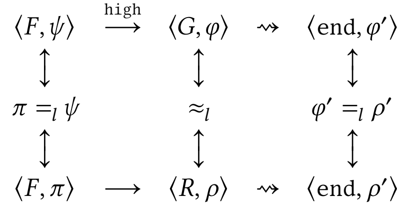


Figure 2: A pictorial representation of the unwinding condition $\mathscr{W}(=_l, \rightsquigarrow, \approx_l)$.

Now that we have gathered all the necessary components, our objective is to demonstrate that a program P does not exhibit interference. In other words, for all possible states σ, the pair $\langle P, \sigma \rangle$ belongs to $\mathscr{W}(=_l, \rightsquigarrow, \approx_l)$. Through our case study, we will illustrate that when this condition is not met, it may lead to MEV interferences.

As demonstrated in our previous work [16], when a program does not belong to an unwinding class, it is possible to define a malicious environment wherein information flows from high to low. This means that by analyzing the program in isolation, we can pinpoint potential hazardous scenarios. In other words, when a program belongs to an unwinding class, no adverse consequences can arise, regardless of the actions taken by the environment. One of the main advantages of the unwinding approach is its ability to identify the specific points within the program that could potentially lead to information flows. As illustrated in Figure 2, such flows occur when high-level transitions are executed.

In the context of MEV interference, this approach aids in identifying the specific dependencies of the contract that require deeper analysis. When scrutinizing these dependencies reveals that no adverse outcomes can occur, it is possible to leverage the concept of downgrading, thereby permitting controlled information flows.

Example 1. In order to provide some more intuition on the meaning of the unwinding condition, let us consider the followning toy example.

$$P \equiv H := 0; \texttt{if}(H > 0)\{L := H\} \ \texttt{else} \ \{\texttt{skip}\}$$

Let H be high level and L be low level. Apparently, when we reach the if test the value of the high level variable is 0, so the else branch is always taken and the value of the high level variable is not revealed. However, this program does not satisfy our unwinding condition as one can observe taking for instance ψ which assigns value 0 to H and π that is $\psi[H/1]$. As a matter of fact, when the if test is reached we have to consider the possibility that another program running in parallel with P has modified the value of H, thus allowing to take the if branch and reveal the value of the variable H to the low level user.

2.3. Downgrading

As noted by numerous researchers, the concept of noninterference can be overly limiting in numerous practical scenarios. Noninterference models the complete absence of information flow from high to low levels. However, in reality, many programs do require some form of

release or *downgrading* of sensitive information (e.g., password checking, information purchase, and spreadsheet computation) [12, 20].

The intuitive concept of downgrading masks certain intricate issues at the implementation level. For example, it raises questions like ”who can perform downgrading,” ”what elements can be downgraded,” ”where downgrading can occur,” and so forth. In our previous work [16], we introduced a collection of high-level expressions and proposed a delimited notion of noninterference that permits the downgrading of such expressions at any stage during the execution.

In this paper, we opt to focus on a notion of downgrading that pertains to the precise point within the execution where the downgrading occurs. This aligns with our approach of using unwinding conditions to pinpoint specific points in the execution of a smart contract where potential MEV interferences may arise. To achieve this, we introduce a function called `downgrade` that maps any arithmetic or boolean expression to itself, while altering its confidentiality level to `low`. Consequently, the operational semantics of program instructions involving `downgrade` are declassified to the low level, and they are not considered as dangerous in the unwinding test.

Example 2. We consider again the program P of Example 1. If we know that revealing to the low level user the value of H is not dangerous, and maybe it is necessary (e.g., the system administrator needs to send to the user a message), then we can modify the program P as follows:

$$P \equiv H := 0; D := \mathtt{downgrade}(H); \mathtt{if}(D > 0)\{L := D\}\ \mathtt{else}\ \{\mathtt{skip}\}$$

If D is a low level variable, the only point in the program where we should check the unwinding condition, is the second assignment instruction. However, since the `downgrade` operator is used, when the assignment is executed a low level transition is performed and the unwinding condition is trivially satisfied.

3. Back to the Bet Contract

We are now ready to model the Bet contract case study within our imperative language. In our model, the Bet contract is defined as the parallel composition of its functionalities, namely:

$$\textsc{Bet_Contract} \equiv \text{co } \textsc{Constructor} \mid \textsc{Bet} \mid \textsc{Win} \mid \textsc{Close} \text{ oc}.$$

Since our language provides only basic instructions and lacks object-oriented features, we exploit the `await` operator to enforce the desired order of execution. For instance, the Constructor should execute first. Additionally, in our code, we denote variables using strings that begin with capital letters, while constants are enclosed in quotation marks.

The Constructor program is responsible for deploying the Bet_Contract. Let us assume a program named Owner initializes the variables InitialPot, Rate, Token, Oracle, and Deadline. Additionally, the variable OracleGetToken contains the returned value of the GetToken program, which is a part of the oracle contract, and it is used to check that the oracle contract allows the exchange between Ether and Token. Another program, Zero, assigns zero values to the variables InitialPot, Rate, Token, Oracle, and Deadline. The Constructor program awaits the initialization of the variables InitialPot, Rate, Token, Oracle, and Deadline. Subsequently, it verifies that the oracle program can exchange Ether and Token. If this is the case, it updates the

variables BetOwner, BetOwnerWalletEther, BetWallet. We highlight the variable BetWallet in green as it is the variable we aim to safeguard from MEV attacks, meaning it will be designated as low-level.

```
1: Program CONSTRUCTOR
2:     await ( InitialPot ≠0 ∧ BetRate ≠0 ∧ Deadline ≠0 ∧ Oracle ≠0 ∧ Token ≠0) do
3:         GETTOKEN
4:     if (Token ≠ 'Ether' ∧ OracleGetToken = ('Ether' , Token) ) then
5:         BetOwner := SenderConstructor;
6:         BetOwnerWalletEther := BetOwnerWalletEther − InitialPot;
7:         BetWallet := BetWallet + InitialPot
8:     else
9:         ZERO
```

The BET program continuously runs until the deadline set by the BET_CONTRACT owner in the variable Deadline is expired. It waits for the Player to be 'NULL' and for PotBet to be non zero. The condition for Player to be 'NULL' ensures that another user is not already placing a bet. Additionally, PotBet must be non-zero, indicating that the PLAYER program has initialized the variables PotBet and SenderBet. If the PotBet is valid, the program updates the variables Player, PlayerWalletEther, and BetWallet.

```
1: Program BET
2:     while (Deadline > BlockNum) do
3:         await (Player = 'NULL' ∧ PotBet ≠0 ) do
4:             skip
5:         if (PotBet = BetWallet) then
6:             Player := SenderBet;
7:             PlayerWalletEther := PlayerWalletEther − PotBet;
8:             BetWallet := BetWallet + PotBet
9:         else
10:            PotBet := 0;
11:            SenderBet := 0
```

Similar to the BET program, the WIN program runs continuously until the deadline expires. It awaits the current player, whose name is stored in the variable Player, to claim victory. This indicates that when the player wishes to claim, they will use another program to modify the value of the variable SenderWin. Upon this action, the WIN program compares the variables BetRate and AmmRateEther. If the latter is greater than the former, the player receives the entire value of the BetWallet. The variable AmmRateEther is highlighted in red because its value is determined and modified within the oracle program (AMM or EXCHANGE in our example). We lack control over its value, making it a potential source of MEV attacks.

```
1: Program WIN
2:     while (Deadline > BlockNum) do
3:         await ( SenderWin = Player ) do
4:             skip
5:         if (BetRate < AmmRateEther) then
6:             PlayerWalletEther := PlayerWalletEther + BetWallet;
```

```
7:          BetWallet := 0
8:      else
9:          SenderWin := 'NULL'
```

It's worth noting that we explicitly designate the variable BetWallet as low-level and AmmRateEther as high-level, while no specific designation is made for the other variables. In this scenario, we can consider them as low-level by default, which is equivalent to solely focusing on potential MEV attacks stemming from the variable AmmRateEther.

At this point, we have all the necessary ingredients to assess whether the program Bet_Contract meets our unwinding condition. Assuming that the only high-level variable is AmmRateEther, we observe that the sole high-level transition is triggered by line 5 of the Win program. When this transition occurs we reach a state which leads to the end of the program with the low level variable BetWallet equal to 0. However, if we modify only the high level variable AmmRateEther prior to the execution of line 5, the program ends with BetWallet being non-zero. This discrepancy is sufficient to infer that Bet_Contract fails to satisfy the unwinding condition, indicating interference from high to low and highlighting a potential source of MEV attacks.

This small example provides insight into the functionality of unwinding conditions, demonstrating their ability to analyze the program of interest independently of the blockchain environment in which it will ultimately operate. Additionally, unwinding conditions offer the benefit of identifying the precise instructions and variables within the code that could potentially be exploited as attack vectors.

However, we cannot draw a conclusion on the reliability of the Bet_Contract solely based on this analysis. It is imperative to delve deeper into the oracle programs to assess whether the potential attacks are indeed feasible. If these attacks are not deemed "realistic", we can then utilize the downgrading mechanism and consider the Bet_Contract as "secure".

Let us now examine the scenario where the oracle program is the AMM contract. Specifically, we will analyze its core functionalities, particularly operations that significantly impact the token's rate. An Automated Market Maker (AMM) is a decentralized finance (DeFi) algorithmic trading protocol that enables the automatic exchange of cryptographic tokens through the use of liquidity pools. Unlike traditional order book-based exchanges, AMMs provide liquidity through these pools, with prices determined algorithmically based on the token ratios within the pool. Automated Market Makers (AMMs) employ mathematical formulas to determine token prices within liquidity pools. The two predominant AMM formulas are the Constant Product Market Maker (CPMM) and the Constant Sum Market Maker (CSMM). In our case study, we use an AMM instance based on CPMM, which is expressed by the formula $K = X * Y$, where X represents the amount of token T1 in the AMM, and Y represents the amount of token T2 in the AMM. This implies that the ratio of token quantities directly influences the price of one token relative to another. Furthermore, the formula ensures that, while the values for tokens X and Y may fluctuate, the value for K remains constant during trades or liquidity deposits.

The program GetRate returns the exchange rate between the two tokens T1 and T2, ensuring the constant product of their quantities as defined earlier. This program computes and provides the value of the variable AmmRateEther, which is used in the Win program within the Bet_Contract. Similarly, in the code, variables designated in red are considered high-

level. Specifically, AmmRateT1 and AmmRateT2 must be high-level because they correspond to the variable AmmRateEther used in the Win program. Furthermore, the variables AmmWalletT1 and AmmWalletT2 also need to be high-level, their significance will become clear upon examining the Swap program.

```
1:  Program GetRate
2:      while true do
3:          await ( T ≠'NULL') do
4:              skip
5:          if (T = 'T1') then
6:              AmmRateT1 := AmmWalletT1 / AmmWalletT2;
7:              T := 'NULL'
8:          else if (T = 'T2') then
9:              AmmRateT2 := AmmWalletT2 / AmmWalletT1;
10:             T := 'NULL'
11:         else
12:             T := 'NULL'
```

The Swap program enables token exchange based on the constant product K. The swapping operation is governed by the values of AmountToSwap and TokenToSwap specified by the sender. If the sender chooses to exchange AmountToSwap of token T1 for token T2, the program initially computes the corresponding amount required for AmountToSwap in T2. Subsequently, it verifies whether AmmWalletT2 holds adequate tokens to satisfy this amount. If it does, the program increases AmmWalletT1 by the specified AmountToSwap and decreases AmmWalletT2 accordingly to maintain the constant stored in the variable K. The same process occurs vice-versa if the sender chooses to swap T2 for T1. We posit the existence of a program, ZeroAMM, which is presumed to initialize the AmountToSwap and TokenToSwap variables to zero.

```
1:  Program Swap
2:      while true do
3:          await (AmountToSwap ≠0 ∧ TokenToSwap ≠'NULL' ) do
4:              skip
5:          K := AmmWalletT1 * AmmWalletT2
6:          if (TokenToSwap = 'T1') then
7:              Y := AmountToSwap * AmmRateT2;
8:              if  (Y < AmmWalletT2) then
9:                  AmmWalletT1 := AmmWalletT1 + AmountToSwap;
10:                 AmmWalletT2 := K / AmmWalletT1;
11:                 ZeroAMM
12:             else
13:                 ZeroAMM
14:         else if (TokenToSwap = 'T2' ) then
15:             Y := AmountToSwap * AmmRateT1;
16:             if  (Y < AmmWalletT1) then
17:                 AmmWalletT2 := AmmWalletT2 + AmountToSwap;
```

```
18:          AmmWalletT1 := K / AmmWalletT2;
19:          ZeroAMM
20:        else
21:          ZeroAMM
```

As far as the levels of the variables are concerned, since AmmRateT1 and AmmRateT2 have to be high level, in order to ensure that the program Swap satisfies the unwinding condition, the variable Y has to be high too (see line 7). This in turn implies that AmmWalletT1 and AmmWalletT2 have to be high (see lines 8,9, 16, 17). Finally, this requires to the variable K to be high (see line 5).

We already observed that line 5 of the Win program causes a flow of information witnessing a possible MEV attack. Now that we possess a deeper understanding of the AMM contract, we can demonstrate the interference resulting from the interaction with the AMM contract, utilizing the LTS notation outlined in Section 2.1. Specifically, we will show how two executions of the same program starting with the same low level variable values can end with different values for the low level variable BetWallet. Let us consider the program:

$$\text{Bet+Amm} \equiv \textbf{co}\ \text{Bet_Contract} \mid \text{Amm_Contract}\ \textbf{oc}$$

In particular, let us assume that we have reached the execution of:

$$\textbf{co}\ (\textbf{co}\ \text{Bet} \mid \text{Win} \mid \text{Close}\ \textbf{oc}) \mid \text{Amm_Contract}\ \textbf{oc}$$

Moreover, the Player has been set to 'BOB'and he has already added his PotBet to the BetWallet. In this context, we designate the T1 token held within the AMM contract to represent Ethereum's native cryptocurrency, Ether. Thus, we denote AmmWalletT1 as AmmWalletEther and AmmRateT1 as AmmRateEther. We focus on a state σ in which the variables have the following values:

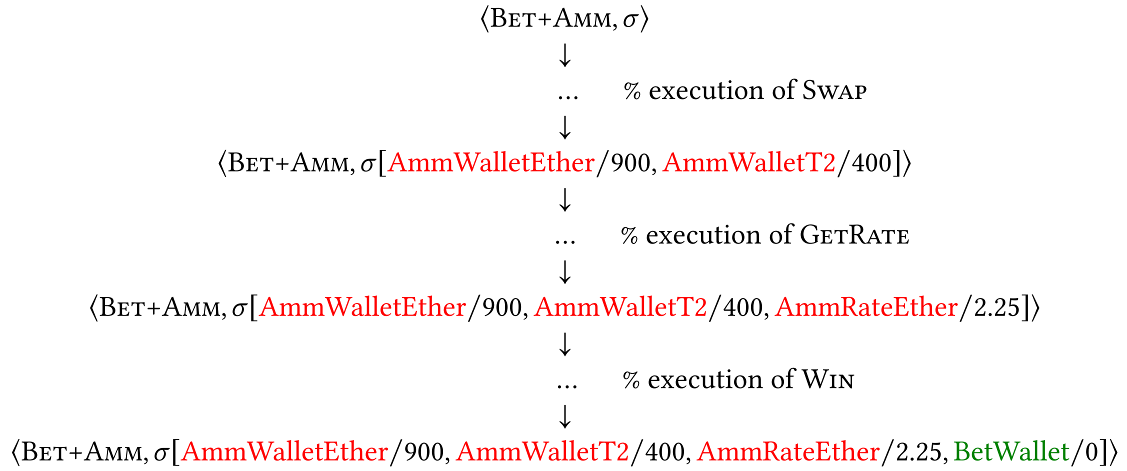
$$\text{AmmWalletEther} = 600;\ \text{AmmWalletT2} = 600;\ \text{AmmRateEther} = 1$$

$$\text{AmountToSwap} = 300;\ \text{TokenToSwap=T2}$$

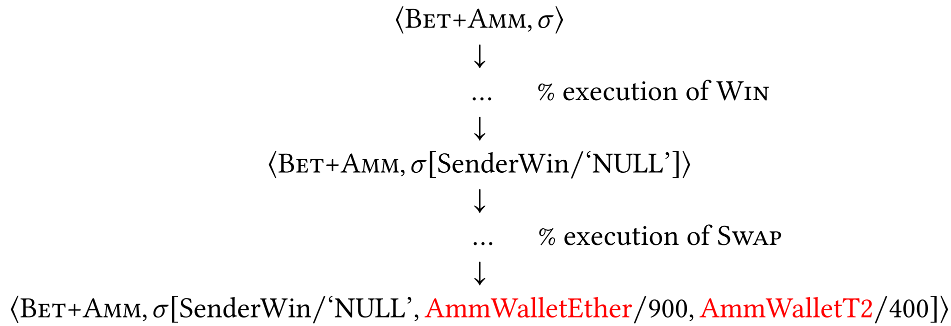
$$\text{BetWallet} = 10;\ \text{BetRate} = 2;\ \text{SenderWin} = \text{'BOB'}$$

If the execution sequence begins with the code of the Swap program, followed by GetRate, which are part of the Amm_Contract, and concludes with the selection of Win, the resulting

outcome is as follows:



Conversely, if Win is executed prior to Swap, and then Swap follows, the outcome differs:



These two executions result in a distinct value for the low level variable BetWallet, indicating a potential MEV attack. In fact, in the second execution the variable BetWallet has not been modified and reaches the end with value 10. Notice that we are not delving into the specific identities of the player and the user initiating the Swap. Nor we are assuming their ability to influence the scheduling order. We are simply observing two different low-level behaviors. Further analysis is required to ascertain the potential transformation of the first execution into a MEV attack. Naturally, if the player of the Bet contract also interacts with the AMM and possesses the authority to dictate transaction order as a miner, they can ensure a favorable outcome, thereby enabling the occurrence of a MEV attack, as highlighted in [18].

Let us now examine the scenario where we leverage the Exchange contract instead of relying on the AMM contract. In this case only the owner has the authority to modify the rate. Below, we model the pivotal functionality of the Exchange contract, wherein the rate is altered.

```
1: Program SetRate
2:     while true do
3:         await (NewRate ≠0 ∧ SenderSetRate = ExchangeOwner) do
4:             skip
5:         ExchangeRate := NewRate;
```

```
6:          NewRate := 0;
7:          SenderSetRate := 'NULL'
```

Let us modify the Win program at line 5 by replacing the variable AmmRateEther with the variable ExchangeRate. Our analysis on the Bet contract is not affected. We still observe that the unwinding condition is not satisfied witnessing a possible MEV attack. However, in this very simple example one can assume that when the player in the Bet contract is different from the owner of the Exchange contract, the player cannot be sure to win the bet. So, we can exploit the downgrading mechanism and modify the Win program as follows.

```
1: Program Win
2:     while (Deadline > BlockNum) do
3:         await ( SenderWin = Player ) do
4:             skip
5:         if (Player ≠ ExchangeOwner) then
6:             CurrentRate:=downgrade(ExchangeRate);
7:         else
8:             CurrentRate:=0;
9:
10:        if (BetRate < CurrentRate) then
11:            PlayerWalletEther := PlayerWalletEther + BetWallet;
12:            BetWallet := 0;
13:        else
14:            Player := 'NULL'
```

In this specific scenario, the noninterference property remains unscathed when utilizing the Exchange contract as a price oracle, given that the adversary lacks the ability to manipulate the exchange rate. With the Exchange contract, only the owner holds the authority to adjust the exchange rate, rendering any attempts by adversaries to influence the exchange contract ineffective in altering the rate. However, it is essential to recognize that within this framework, there exists a potential information flow through interactions with the Exchange contract, particularly when the owner of the Exchange contract coincides with the owner or player of the Bet contract. This introduces the possibility of interference. However, interference does not arise when the player in the Bet contract differs from the owner of the Exchange contract. To differentiate this secure scenario from the potentially risky one described earlier, we rely on the Downgrading functionality embedded in our language.

The code implementation for this example encompasses smart contracts for Bet, AMM, and Eexchange functionalities. Furthermore, we've developed a USDC ERC20 token to enable a hands-on exchange experience with our custom token. The complete implementation in solidity can be found on GitHub via the following link: code repository.

4. Conclusion

In this paper, we explore the application of unwinding conditions to analyze Maximal Extractable Value (MEV) within the context of the Bet contract. Our investigation utilizes a simple imperative concurrent language to model the Bet contract, shedding light on scenarios where specific

conditions impact MEV noninterference. This underscores potential vulnerabilities within decentralized finance (DeFi) services. This is a first step, while an in depth analysis of the relationships between unwinding conditions and MEV is left as future work.

Comparing our approach with the contribution of Bartoletti et al. in[18] (see also Appendix A), our focus lies in formalizing noninterference through unwinding conditions. By employing unwinding conditions, we abstract from the environment, enabling us to disregard the specific implementations of possible smart contracts interacting with the one under examination, unlike in [18]. Clearly this is an advantage from the computational point of view, since it allows us to analyse smaller system. Moreover, this approach facilitates the identification of critical points within the program that may lead to information flows, thereby enhancing the security analysis of smart contracts. On the other side, a second stage analysis of the dependencies is necessary in order to discard unrealistic sources of attacks.

Furthermore, the Bet contract case study presented in this paper illustrates the practical application of our methodology in identifying and mitigating MEV attacks. Overall, our work contributes to the comprehension and mitigation of MEV vulnerabilities in smart contracts within the realm of decentralized finance. As future work we also plan to apply our framework.

As future work we also plan to define our framework on fragments of languages for smart contracts, such as solidity, in order to avoid possible discrepancies introduced relying on an intermediate language. On the one hand, this is a difficult task due to the richness of languages for smart contracts. On the other hand, object oriented features would avoid us the use of programming tricks, such as the use of the away operator, for avoiding unwanted interleaving behaviors.

Acknowledgments

This work has been partially supported by the Project PRIN 2020 "Nirvana - Noninterference and Reversibility Analysis in Private Blockchains", and by the project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU.

References

[1] ethereum.org: what-is-defi?, https://ethereum.org/en/defi/#what-is-defi, 2024. Accessed: jan 13, 2024.

[2] L. Gudgeon, S. Werner, D. Perez, W. J. Knottenbelt, Defi protocols for loanable funds: Interest rates, liquidity and market efficiency, in: AFT '20: 2nd ACM Conference on Advances in Financial Technologies, ACM, 2020, pp. 92–112. doi:10.1145/3419614.3423254.

[3] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, W. J. Knottenbelt, Sok: Decentralized finance (defi), in: Proceedings of the 4th ACM Conference on Advances in Financial Technologies, AFT 2022, ACM, 2022, pp. 30–46. doi:10.1145/3558535.3559780.

[4] J. Xu, K. Paruch, S. Cousaert, Y. Feng, Sok: Decentralized exchanges (DEX) with automated market maker (AMM) protocols, ACM Comput. Surv. 55 (2023) 238:1–238:50. doi:10.1145/3570639.

[5] S. Kitzler, F. Victor, P. Saggese, B. Haslhofer, A systematic investigation of defi compositions in ethereum, in: Financial Cryptography and Data Security. FC 2022 International Workshops - CoDecFin, DeFi, Voting, WTSC, volume 13412 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 272–279. doi:10.1007/978-3-031-32415-4_18.

[6] L. Zhou, X. Xiong, J. Ernstberger, S. Chaliasos, Z. Wang, Y. Wang, K. Qin, R. Wattenhofer, D. Song, A. Gervais, Sok: Decentralized finance (defi) attacks, in: 44th IEEE Symposium on Security and Privacy, SP 2023, IEEE, 2023, pp. 2444–2461. doi:10.1109/SP46215.2023.10179435.

[7] P. Daian, S. Goldfeder, T. Kell, Y. Li, X. Zhao, I. Bentov, L. Breidenbach, 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 2020, IEEE, 2020, pp. 910–927. doi:10.1109/SP40000.2020.00040.

[8] D. Bernhardt, B. Taub, Front-running dynamics, J. Econ. Theory 138 (2008) 288–296. doi:10.1016/J.JET.2007.05.005.

[9] K. Li, S. Guan, D. Lee, Towards understanding and characterizing the arbitrage bot scam in the wild, Proc. ACM Meas. Anal. Comput. Syst. 7 (2023) 52:1–52:29. doi:10.1145/3626783.

[10] J. A. Goguen, J. Meseguer, Security policies and security models, in: 1982 IEEE Symposium on Security and Privacy, 1982, IEEE Computer Society, 1982, pp. 11–20. doi:10.1109/SP.1982.10014.

[11] R. Focardi, S. Rossi, Information flow security in dynamic contexts, J. Comput. Secur. 14 (2006) 65–110. doi:10.3233/JCS-2006-14103.

[12] S. Crafa, S. Rossi, Controlling information release in the pi-calculus, Inf. Comput. 205 (2007) 1235–1273. doi:10.1016/J.IC.2007.01.001.

[13] J. Hillston, A. Marin, C. Piazza, S. Rossi, Persistent stochastic non-interference, Fundam. Informaticae 181 (2021) 1–35. doi:10.3233/FI-2021-2049.

[14] A. Bossi, C. Piazza, S. Rossi, Unwinding conditions for security in imperative languages, in: Logic Based Program Synthesis and Transformation, 14th International Symposium, LOPSTR 2004, volume 3573 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 85–100. doi:10.1007/11506676_6.

[15] A. Bossi, R. Focardi, C. Piazza, S. Rossi, Verifying persistent security properties, Comput. Lang. Syst. Struct. 30 (2004) 231–258. doi:10.1016/J.CL.2004.02.005.

[16] A. Bossi, C. Piazza, S. Rossi, Compositional information flow security for concurrent programs, J. Comput. Secur. 15 (2007) 373–416. doi:10.3233/JCS-2007-15303.

[17] K. Babel, P. Daian, M. Kelkar, A. Juels, Clockwork finance: Automated analysis of economic security in smart contracts, in: 44th IEEE Symposium on Security and Privacy, SP 2023, IEEE, 2023, pp. 2499–2516. doi:10.1109/SP46215.2023.10179346.

[18] M. Bartoletti, R. Marchesin, R. Zunino, Defi composability as MEV non-interference, CoRR abs/2309.10781 (2023). doi:10.48550/ARXIV.2309.10781.

[19] G. R. Andrews, Foundations of Multithreaded, Parallel, and Distributed Programming, Addison-Wesley, 2000.

[20] A. Marin, C. Piazza, S. Rossi, *D_PSNI*: Delimited persistent stochastic non-interference, Theor. Comput. Sci. 884 (2021) 116–135. doi:10.1016/J.TCS.2021.08.007.

A. MEV and Local MEV in [18]

In [18], the author delineates two notions of MEV. The first, termed global MEV, relies on the entire blockchain state S to determine the value of MEV extracted from a contract Δ. This criterion introduced by Babel et al. in [17], referred to as ε-composability, asserts that contracts Δ are composable in a blockchain state S if adding Δ to S does not afford the adversary a "significantly higher" Maximal Extractable Value (MEV). Formally, denoting by MEV(S) the maximal value adversaries can extract from S, and with $S|\Delta$ representing the blockchain state S extended with contracts Δ, the composability criterion of Babel et al. is expressed as:

$$\Delta \text{ is } \varepsilon\text{-composable in } S \text{ if } \text{MEV}\,(S|\Delta) \leq (1+\varepsilon)\,\text{MEV}\,(S) \tag{1}$$

where ε parameterizes the "not significantly higher" condition above. However, this approach has limitations as elucidated in [18]. For instance, utilizing the MEV of the entire state S as a baseline for comparison complicates the interpretation of the concrete security guarantee of ε-composability.

The second notion of MEV, introduced by Bartoletti et al., is Local MEV, termed MEV non-interference. This concept measures the maximal economic loss that adversaries can induce on a given set of contracts, unlike global MEV, which applies solely to the entire blockchain state. By adapting the definition of non-interference property to the DeFi setting, they stipulate that the MEV extractable from Δ (the public output) should remain unaffected by the dependencies of Δ (the private inputs). Formally, MEV non-interference holds when the MEV extractable from the contract accounts in Δ(i.e., $\dagger\Delta$) using any contract in $S \mid \Delta$ is exactly the same MEV that can be extracted using only the contracts in Δ. They introduce the following definition:

Definition 3. (MEV non-interference). A state S is MEV non-interfering with Δ, when

$$\text{MEV}(S|\Delta, \dagger\Delta) = \text{MEV}^{\dagger}{}_{\Delta}(S|\Delta, \dagger\Delta). \tag{2}$$