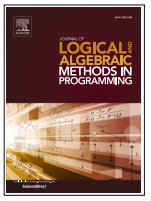
J. Log. Algebraic Methods Program. 152 (2027) 101171



Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

journal homepage: www.elsevier.com/locate/jlamp



A formal framework for the economic security of DeFi compositions

Massimo Bartoletti [a,*], Riccardo Marchesin [b], Roberto Zunino [b]

[a] University of Cagliari, Cagliari, Italy
[b] University of Trento, Trento, Italy

ARTICLE INFO

Keywords:
Smart contracts
Blockchain
Decentralized applications
Formal methods
Verification

ABSTRACT

Decentralised Finance (DeFi) protocols are usually built by composing a variety of smart contracts. Although composability is a key driver of DeFi innovation, it also introduces security risks: adversaries may exploit interactions between newly deployed contracts and the existing blockchain state to cause economic losses. We introduce a formal security notion for DeFi composability, called *MEV non-interference*, which requires that interactions with the existing blockchain state do not increase the Maximal Extractable Value (MEV) obtainable from a set of newly deployed contracts. To formalise this notion, we define *local MEV*, a new measure of economic attacks that quantifies the losses incurred by a given set of victim contracts. We study MEV non-interference under two adversarial models, corresponding to bounded and unbounded adversarial wealth, and establish sufficient conditions and locality principles that enable modular reasoning about secure composability. We apply our framework to representative DeFi compositions including exchanges, automated market makers, options, lending pools, routers, and arbitrage contracts, and show that it distinguishes secure compositions from vulnerable ones. Our results provide a formal foundation for analysing the economic security of DeFi systems built through smart contract composition.

1. Introduction

Decentralised Finance (DeFi) is often described as “money Lego” because it enables the construction of complex financial services through the composition of simpler building blocks [1,2]. Recent empirical analyses of the DeFi ecosystem confirm that DeFi protocols are deeply interconnected in practice, giving rise to highly intricate dependency structures and interactions [3,4]. While such composability is a key driver of the success of DeFi, it also introduces significant security challenges: adversaries can exploit unintended interaction among protocols to obtain an economic profit to the detriment of honest users [5,6]. These risks are further exacerbated by emerging platforms that allow users to seamlessly create arbitrary compositions of DeFi protocols [7]. This raises a fundamental and still largely unanswered question: *under which conditions is a DeFi composition secure?*

Despite the clear practical relevance of this question, research on DeFi composability remains surprisingly limited. The first notion of secure DeFi composition in the scientific literature is the one introduced by Babel, Daian, Kelkar and Juels in their seminal “Clockwork finance” paper [8]. The focus of this work is on attacks in which adversaries exploit newly deployed contracts to increase their profit opportunities. Accordingly, their notion of composability requires that augmenting a blockchain state S with a set of contracts Δ should not allow adversaries a to extract a substantially larger Maximal Extractable Value (MEV). In formulae, letting

* Corresponding author.
E-mail address: bart@unica.it (M. Bartoletti).

https://doi.org/10.1016/j.jlamp.2026.101171
Received 3 June 2026; Received in revised form 10 September 2026; Accepted 14 September 2026
Available online 23 September 2026
2352-2208/© 2026 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

$\mathrm{MEV}(S)$ denote the maximal value that adversaries can extract from S, and writing $S \mid \Delta$ for the blockchain state S extended with the contracts Δ, their criterion is expressed as:

$$\Delta \text{ is } \varepsilon\text{-composable in } S \quad \text{iff} \quad \mathrm{MEV}(S \mid \Delta) \leq (1+\varepsilon)\,\mathrm{MEV}(S) \tag{1}$$

where ε parameterises the "not substantially larger" condition above. For example, let Δ be a contract that allows users to bet on the future price of a token, using an Automated Market Maker (AMM) in S as a price oracle. The key issue is that the AMM prices are not fixed: they are determined by the current token reserves held by the AMM and therefore fluctuate in response to trades. In particular, large trades can induce substantial price fluctuations. If the adversary has sufficient capital in S, criterion (1) would correctly classify Δ as *not* 0-composable in S. Indeed, the adversary can strategically manipulate the AMM price by executing large trades, thereby creating artificial price fluctuations that allow them to systematically win the bet. As a result, the adversary can extract strictly more MEV from $S \mid \Delta$ than from S alone.

Limitations of ε-composability. We argue that the ε-composability has some limitations.

First, using the MEV of the *entire* blockchain state S as a baseline for comparison makes the security guarantee provided by ε-composability difficult to interpret. One might naturally conclude that deploying a contract Δ in S is safe after establishing that Δ is 0-composable with S. However, 0-composability only guarantees that the total MEV of the extended state $S \mid \Delta$ does not exceed that of S; it does not constrain *where* this MEV is extracted from. To illustrate, suppose that the total value of tokens in Δ is equal to the MEV of S, and that extracting value from Δ eliminates the MEV opportunities previously available in S. Then Δ is still 0-composable with S, even though an attack on $S \mid \Delta$ may drain the entire value of Δ (see Example 18 for a concrete scenario). From the perspective of the designer or users of Δ, such a composition would hardly be considered secure.

A second drawback of using the *global* MEV as a baseline is that (1) classifies as *not* composable contracts whose MEV is intentional and does not arise from any interaction with the rest of the system. Consider, for instance, an airdrop contract Δ that allows anyone to withdraw its entire balance. Despite being completely independent from the rest of the system, Δ is not 0-composable with any S, since deploying it necessarily increases the total MEV of the system, i.e., $\mathrm{MEV}(S \mid \Delta) > \mathrm{MEV}(S)$. One could attempt to recover composability by choosing a sufficiently large value of ε, but this is both cumbersome and conceptually unsatisfactory. In the extreme case where $\mathrm{MEV}(S) = 0$, the contract is not ε-composable for any $\varepsilon \geq 0$, regardless of its complete independence from the rest of the system.

Finally, the reliance on global MEV poses both algorithmic and usability concerns. From a computational standpoint, evaluating (1) requires reasoning about a property of the entire blockchain state, which may be prohibitively large in practice. From a mitigation standpoint, when ε-composability fails because additional MEV can be extracted from contracts already present in S, it is unclear how the designer of Δ should react. The vulnerable contracts reside in the pre-existing state and are therefore outside the control of the deployer. Consequently, the definition may identify a composability problem without providing actionable guidance on how to address it.

Secure composability as non-interference. The limitations of ε-composability suggest the need for composability notions that do not rely on the global MEV of the blockchain state.

Our starting point is the notion of *non-interference*, which has been extensively studied in information-flow security since the 1980s [9–13]. In its classical form, non-interference requires that adversaries interacting with a software system cannot observe private data. More precisely, the property holds when public outputs (that can be observed by adversaries) are not affected by private inputs.

We adapt this principle to the setting of smart contracts composition, ending up with a notion that we call *MEV non-interference*. Intuitively, the contracts Δ being deployed should not become more vulnerable merely because they are composed with a larger blockchain state. In other words, the maximal economic loss that adversaries can inflict on Δ should only depend on the interactions of the adversary with Δ, and not with the rest of the system. Under this interpretation, the interactions between the adversary and the surrounding system S play the role of the private inputs in classic non-interference, while the economic loss suffered by Δ plays the role of the public outputs. MEV non-interference therefore requires that interacting with contracts in S does not enable adversaries to cause more loss on Δ than they could achieve by interacting with Δ alone.

We argue that MEV non-interference captures the security property that protocol designers are naturally interested in when deploying new contracts. By focusing on the *local* MEV extractable from Δ, our notion overcomes the above-mentioned limitations of ε-composability. First, contracts whose MEV is intentional are not necessarily classified as insecure. For example, deploying the above-mentioned airdrop contract in any S is considered secure by MEV non-interference. Second, MEV non-interference admits local reasoning: under suitable conditions, one can establish the composability of Δ by reasoning only about Δ and its dependencies, without analysing the entire blockchain state.

Contributions. This paper introduces a formal framework for reasoning about the secure composability of smart contracts. We summarise our main contributions as follows:

- **MEV non-interference.** We adapt the classical notion of non-interference from information-flow security to the smart contracts setting, obtaining a novel security criterion for protocol composition. To the best of our knowledge, this is the first application of non-interference principles to the study of smart contracts composability.
- **Local MEV.** We introduce *local MEV*, a new measure of economic attacks that quantifies the maximal loss that adversaries can inflict on a given set of victim contracts. In contrast to classical MEV [8,14], which is defined for an entire blockchain state, local

Table 1
Summary of notation.

$\mathtt{A}, \mathtt{B}$	User accounts	$\mathcal{A}, \mathcal{B}$	Sets of [user|contract] accounts
$\mathtt{C}, \mathtt{D}$	Contract accounts	$\mathtt{a}, \mathtt{b}$	User|contract accounts
C, D	Sets of contract accounts	$deps(S)$	Dependencies of S
$\mathtt{T}, \mathtt{T}'$	Token types	$\$\mathbf{1}_{\mathtt{T}}$	Price of $\mathtt{T}$
$\mathtt{X}, \mathtt{X}'$	Transactions	$\vec{\mathcal{X}}$	Sequence of transactions
S, S'	Blockchain states	$\dagger S$	Contract accounts in S
W, W'	User accounts states	$\omega_{\mathcal{A}}(S)$	Wallets of $\mathcal{A}$ in S
Σ, Γ, Δ	Contract accounts states	$\$_{\mathcal{A}}(S)$	Wealth of $\mathcal{A}$ in S
$\mathcal{M}$	Set of adversaries	$\gamma_{\mathcal{A}}(S, \vec{\mathcal{X}})$	$\mathcal{A}$'s gain upon firing $\vec{\mathcal{X}}$ in S

MEV focuses on the contracts whose security is under analysis. This shift from a global to a local perspective is the key ingredient of our approach.

- **Two adversarial models.** We define two variants of local MEV. The first, $\text{MEV}(S, C)$, models adversaries whose wealth is bounded by the available capital. The second, $\text{MEV}^{\infty}(S, C)$, models adversaries with unbounded capital, capturing attacks enabled by mechanisms such as flash loans [15].
- **MEV non-interference.** Building on local MEV, we introduce two notions of secure composability: $S \not\rightarrow \Delta$ for bounded-wealth adversaries and $\Gamma \not\rightarrow^{\infty} \Delta$ for unbounded-wealth adversaries (where Γ denotes a system of contracts obtained by removing users' wallets from the blockchain state). Intuitively, these properties require that composing Δ with an existing blockchain state does not create additional opportunities to extract value from Δ.
- **Local reasoning principles.** We establish locality results showing that both local MEV and MEV non-interference can be analysed by considering only a suitable fragment of the blockchain state. For local MEV, we identify conditions under which the maximal loss suffered by the victim contracts depends only on those contracts and their dependencies (Lemmas 4 and 12 for bounded- and unbounded-wealth adversaries, respectively). Building on these results, we prove that MEV non-interference is preserved for suitable state narrowings (Theorems 15 and 18). Intuitively, we can remove from the state the non-dependencies of Δ, provided that the contracts in Δ are sender-agnostic, i.e. their behaviour is uniform with respect to their callers. For bounded-wealth adversaries, we additionally require that the removed contracts do not induce token flows to Δ or its dependencies. These locality principles provide both conceptual and algorithmic advantages over approaches based on global MEV.
- **Sufficient conditions for secure composability.** We establish practical sufficient conditions for MEV non-interference (Theorems14 and 17). A key ingredient is *observational invariance*, a property ensuring that adversarial interactions with the existing blockchain state cannot influence what the newly deployed contracts observe. Combined with suitable conditions on token flows, observational invariance guarantees that composing a contract with an existing system does not create additional opportunities for adversarial value extraction.
- **Analysis of representative DeFi compositions.** We validate our framework on a collection of representative DeFi compositions, including exchanges, AMMs, binary options, lending pools, swap routers, and arbitrage protocols (Table 2). Using the sufficient conditions and locality principles developed in the paper, we establish secure composability results for several classes of protocol compositions and identify concrete scenarios in which composability fails because of exploitable dependencies, token flows, or manipulable price oracles. The resulting classification highlights the practical usefulness of MEV non-interference as a tool for analysing real-world DeFi ecosystems.

We provide detailed proofs of our statements either in the main text or in the appendices.

2. Blockchain model

We fix a model of account-based blockchains and smart contracts inspired by Ethereum. To keep our theory general, we abstract from some of Ethereum's peculiar features, such as its underlying consensus protocol, gas mechanism, and EVM bytecode, just focussing on the high-level behaviour of smart contracts. We discuss the abstractions of our model in Section 10.

Tokens. We assume a set $\mathbb{T}$ of *token types* ($\mathtt{T}, \mathtt{T}', \ldots$), which represent the types of exchanged (fungible) assets. While Ethereum handles its native crypto-currency (the $\mathtt{ETH}$) differently from the custom assets (smart contracts implementing the ERC-20 interface), we provide a uniform treatment of assets, handling both $\mathtt{ETH}$ and other token types with the same set of operators.

Accounts. We assume a countably infinite set $\mathbb{A}$ of *accounts*, partitioned into:

- *user accounts* $\mathtt{A}, \mathtt{B}, \ldots \in \mathbb{A}_u$, which represent the active entities who can initiate *transactions* (the so-called "externally owned" accounts in Ethereum);
- *contract accounts* $\mathtt{C}, \mathtt{D}, \ldots \in \mathbb{A}_c$, which represent the passive entities that can react to transactions initiated by user accounts.

We use $\mathtt{a}, \mathtt{b}, \ldots$ as meta-variables ranging over both user accounts and contract accounts. We designate a subset $\mathcal{M}$ of user accounts as *adversaries*, i.e., users with the ability to influence the selection and ordering of transactions in the blockchain. In the Ethereum ecosystem, this subset includes, e.g., block proposers and MEV searchers [16]. Although the analysis of a specific contract may depend

on the particular choice of $\mathcal{M}$, there exist natural and robust choices — such as a finite set of fresh user accounts not already appearing in the contract code or state — for which the resulting analysis is insensitive to this choice.

Wallets. A *wallet* $w \in \mathbb{T} \to \mathbb{N}$ is a finite map from token types to non-negative integers, quantifying the amount of tokens held by an account. We represent a wallet as a comma-separated list of associations between a token type $\mathtt{T}$ and its amount n, written as $n : \mathtt{T}$. We denote by $+$ the pointwise summation of wallets.

Blockchain states. A blockchain state S is a finite map from accounts to their respective states. The state of a user account is a wallet, while that of a contract account is a pair (w, σ), where w is the contract's wallet and σ is its persistent storage, represented as a key-value store. We denote by:

- $\dagger S$ the set of contract accounts in S, i.e. $\dagger S = (\mathrm{dom}\, S) \cap \mathbb{A}_c$;
- $\omega_{\mathcal{A}}(S)$ the summation of the wallets of accounts $\mathcal{A}$ in S.

Since the primary focus of this paper is the composability of contracts, it is often necessary to manipulate blockchain states so as to add or remove accounts. This, in turn, requires a suitable notation for representing the individual components of a state, as well as for constructing and deconstructing states in a systematic manner. For this purpose, we adopt the following notation:

- We denote by $\mathtt{A}[w]$ the state of a user account $\mathtt{A}$ whose wallet is w.
- We denote by $\mathtt{C}[w, \sigma]$ the state of a contract account $\mathtt{C}$ whose wallet is w and storage is σ. A storage is represented as a comma-separated list of associations between keys and values, each written as $\mathtt{k} = v$.
- We denote by $S \mid S'$ the composition of two blockchain states S and S'. Such composition is defined whenever $\mathrm{dom}\, S$ and $\mathrm{dom}\, S'$ are disjoint — namely, a blockchain state cannot contain multiple states for the same account.
- We use $W, W', \ldots$ to denote states comprising only user accounts.
- We use $\Sigma, \Gamma, \Delta, \ldots$ to denote states comprising only contract accounts.

Contracts. For generality, rather than assuming a specific syntax for the contract language, we model contracts abstractly as public APIs whose methods have the following behaviour:

1. receive parameters from the caller. In particular, the parameters include the immediate caller and the transaction initiator (these accounts may coincide);
2. perform a finite sequence of *effects* drawn from the following kinds:
   - abort execution when specified conditions are not met;
   - read and update the contract storage;
   - send tokens from the contract to another account. Its behaviour depends on whether the recipient is a user or a contract account:
     - If the recipient is a user account and the sender's wallet contains a sufficient amount of tokens, then the tokens are immediately transferred to the recipient, possibly creating the account if it does not yet exist in the blockchain state.
     - If the recipient is a contract account, then the operation is interpreted as a mere *approval* of the transfer, while the actual transfer is triggered by an explicit *receive* action of the recipient.
   - receive tokens from the caller, provided that it has previously approved the transfer;
   - invoke methods of the same or of other contracts;
3. return a value to the caller.

Note that the aforementioned contract capabilities do not include the ability to directly inspect the state of external accounts, including their wallets. This, however, is not a practical limitation. Dependence on a user's wallet can be enforced by requiring the caller of a method to provide tokens to the contract, while dependence on the state of another contract can be realised by querying that state through its public getter methods. For simplicity, we also forbid direct token transfers between user accounts. This restriction entails no loss of generality, since such transfers can always be mediated through suitable contracts.

Contract dependencies. We say that a contract $\mathtt{C}$ is *called by* $\mathtt{D}$ if some method of $\mathtt{D}$ invokes a method of $\mathtt{C}$. We denote by $\sqsubseteq$ the reflexive and transitive closure of this relation. Given a set of contracts C, we denote by $deps(C)$ the set of *dependencies* of C, namely, the set of contracts called by some contract in C. Formally:

$$deps(C) \;=\; \left\{ \mathtt{C}' \;\middle|\; \exists \mathtt{C} \in C.\, \mathtt{C}' \sqsubseteq \mathtt{C} \right\}$$

We write $deps(S)$ as a shorthand for $deps(\dagger S)$, i.e. the dependencies of the contracts occurring in S.

Well-formedness. We say that a blockchain state $S \;=\; \mathtt{a}_1[\cdots] \mid \cdots \mid \mathtt{a}_n[\cdots]$ is *well-formed* if

- $\sqsubseteq$ is a partial order and
- $\forall i, \mathtt{D}.\ \mathtt{D} \sqsubseteq \mathtt{a}_i \implies \exists j \le i.\ \mathtt{D} = \mathtt{a}_j$.

Condition 1 requires that contracts can only call previously deployed contracts or themselves. Condition 2 means that all the dependencies of the contracts in S are included in S, and if a contract $\mathtt{C} = \mathtt{a}_i$ calls $\mathtt{D}$, then $\mathtt{D}$ appears on the left of $\mathtt{C}$ in the enumeration, i.e. $j \leq i$. Note that if S is well-formed, then:

- S is closed under dependencies, i.e. $deps(S) \subseteq \dagger S$;
- mutual calls between contracts are forbidden;
- when deconstructing S into $S_0 \mid S_1$, the leftmost component S_0 is guaranteed to be well-formed, but the rightmost component S_1 is *not*, since some of the dependencies of S_1 may be in S_0.

Hereafter, we consider well-formed states up-to permutations of their individual components, provided that the well-formedness condition is respected. Unless otherwise stated, in our statements we implicitly assume that the blockchain states mentioned (in hypothesis or thesis) are well-formed.

We anticipate that many typical DeFi compositions enjoy our well-formedness condition (see Section 7). This condition imposes regular structure on contract dependencies, which we exploit to obtain several results. Ill-formedness mostly arises from dynamically supplied addresses, which may allow a contract to reference another contract deployed later, possibly creating a circular dependency. Remarkably, several key results — including Theorems 14 and 17, which give sufficient conditions for MEV non-interference — continue to hold even in the absence of well-formedness.

Transactions. We model blockchains as reactive systems whose behaviour is specified as a deterministic transition relation $\rightarrow$ between blockchain states, and transitions are labelled by *transactions* $\mathtt{X}, \mathtt{X}', \ldots$ sent by users. We denote by:

$$\mathtt{A} : \mathtt{C}.\mathtt{f}(\mathtt{args})$$

a transaction sent by $\mathtt{A}$, where $\mathtt{C}$ is the called contract, $\mathtt{f}$ is the called method, and $\mathtt{args}$ is the list of actual parameters. Given $\mathtt{X} = \mathtt{A} : \mathtt{C}.\mathtt{f}(\mathtt{args})$, we write $callee(\mathtt{X})$ for the called contract $\mathtt{C}$. Note that, although a transaction must originate from a user account, it can trigger additional method calls where the caller is a contract account. In general, a transaction can involve a sequence of calls of the form:

$$\mathtt{A}_0 : \mathtt{D}_0.\mathtt{f}_0(\mathtt{args}_0)\ \ \mathtt{C}_1 : \mathtt{D}_1.\mathtt{f}_1(\mathtt{args}_1)\ \ \mathtt{C}_2 : \mathtt{D}_2.\mathtt{f}_2(\mathtt{args}_2)\ \ \cdots\ \ \mathtt{C}_n : \mathtt{D}_n.\mathtt{f}_n(\mathtt{args}_n)$$

We refer to the method calls $\mathtt{C}_i : \mathtt{D}_i.\mathtt{f}_i(\mathtt{args}_i)$ as *internal* calls, to their callers $\mathtt{C}_i$ as *senders*, and to $\mathtt{A}_0$ as the *origin*. Each method $\mathtt{f}_i$ in this sequence can refer to the origin and to the sender; note that these two accounts coincide only for $i = 0$, while the senders for $i > 0$ are contract accounts.

We assume that invalid transactions are *reverted*, namely:

$$S \xrightarrow{\mathtt{X}} S \qquad \text{if } \mathtt{X} \text{ is not valid in } S$$

There are several possible causes of invalidity, such as attempting to call a non-existent method, triggering an abort, performing an undefined operation (e.g., division by zero), or attempting a token transfer without sufficient balance.

```
interface IPriceOracle {
  // exchange rate to sell ti and buy to (multiplied by 1e6)
  function getRate(token ti, token to) external returns (uint);
}

contract PriceOracle is IPriceOracle {
  mapping (token => uint) prices;
  mapping (token => bool) isPriced;
  address owner;
  uint public immutable fee;

  constructor(uint fee_) { owner=msg.sender; fee=fee_; }

  function setPrice(token t, uint p) public {
    require (msg.sender==owner); // only the owner can set the price
    isPriced[t]=true;
    prices[t]=p;
  }

  function getRate(token ti, token to) public returns(uint) {
    require (isPriced[ti] && isPriced[to]);
    ETH.receive(fee); // transfer fee:ETH from msg.sender to this
    return (prices[ti] * 1_000_000)/prices[to];
  }
}
```

Listing 1. A price oracle contract.

```
interface IExchange is IPriceOracle {
  // list of swappable tokens
  function getTokens() external view returns (token[] memory);
  // receive x:ti & send >=ymin:to; revert if pair is not swappable
  function swap(token ti, uint x, token to, uint ymin) external returns (uint);
}

contract Exchange is IExchange {
  IPriceOracle p; token immutable t0; token immutable t1;

  constructor(IPriceOracle p_, token t0_, token t1_) { p=p_; t0=t0_; t1=t1_; }

  function getTokens() public view returns (token[] memory) {
    token[] memory ts = new token[](2); ts[0]=t0; ts[1]=t1; return ts;
  }

  function getRate(token ti, token to) public view returns (uint) {
    return p.getRate(ti,to);
  }

  function swap(token ti, uint x, token to, uint ymin) public returns (uint) {
    require ((ti==t0 && to==t1) || (ti==t1 && to==t0));
    ti.receive(x); // transfer x:ti from msg.sender to this
    ETH.receive(p.fee());  // transfer fee from sender to this
    ETH.send(p.fee(), p);  // approve fee transfer to oracle
    uint y = (x * p.getRate(ti,to))/1_000_000;
    require (y>=ymin);
    to.send(msg.sender,y); // transfer y:to to msg.sender
    return y;
  }
}
```

Listing 2. A simple exchange contract handling two token types.

Contract language. In our examples, we specify contracts in a Solidity-like language with primitive token types. Namely, rather than implementing tokens as ERC-20 contracts, our language extends Solidity with a base type token to represent token types. Each token type T supports the following operations:

- T.send(rcv,n) has different behaviours depending on whether the recipient `rcv` is a user or a contract account. If `rcv` is a user account, then the operation transfers `n:T` from the contract to `rcv`. Instead, if `rcv` is a contract account, the operation authorizes the contract `rcv` to receive up to `n:T` from the caller's wallet. We assume that a subsequent send with the same token and receiver overwrites the previous one, effectively cancelling it;
- T.receive(n) transfers `n:T` from the immediate caller to the contract. The operation reverts if the caller has not previously approved the transfer, or if its wallet does not contain a sufficient balance of tokens `T`.

Except for the token type and its operations, our code snippets adhere to Solidity.[1] We will be a bit sloppier only in the technical toy examples, where for brevity we will omit, e.g., the function visibility and mutability modifiers.

Example 1. The contract `PriceOracle` in Listing 1 has two methods: `setPrice` allows the owner to set the price of a given token type (taking as unit of measure some reference token, e.g. the `ETH`), and `getRate` allows anyone to read the exchange rate between two token types, provided that a fee is paid to the contract. The contract `Exchange` in Listing 2 allows anyone to pay tokens of type `t0` in exchange of tokens of type `t1` (and vice versa). The exchange rate is obtained through a price oracle, fixed when `Exchange` is deployed.

Consider a transaction $\mathtt{X} = \mathtt{A} : \mathtt{Exchange.swap}(\mathtt{T}_1, 10, \mathtt{T}_2, 0)$ and the state:

$$\begin{aligned} S \;&= \mathtt{A}[10:\mathtt{T}_1, 0:\mathtt{T}_2, 1:\mathtt{ETH}] \mid \mathtt{B}[0:\mathtt{ETH}] \\ &\mid \mathtt{PriceOracle}[0:\mathtt{ETH}, \mathtt{owner}=\mathtt{B}, \mathtt{fee}=1, \mathtt{prices}=\{\mathtt{T}_1=3, \mathtt{T}_2=5\}, \ldots] \\ &\mid \mathtt{Exchange}[0:\mathtt{T}_1, 100:\mathtt{T}_2, \mathtt{p}=\mathtt{PriceOracle}, \mathtt{t0}=\mathtt{T}_1, \mathtt{t1}=\mathtt{T}_2] \end{aligned}$$

Since transactions are signed by their senders, we do not require them to explicitly approve token transfers (as in the Move language [17]). Firing X in S triggers an internal call $\mathtt{Exchange} : \mathtt{PriceOracle.getRate}(\mathtt{T}_1, \mathtt{T}_2)$, overall causing a state transition from S to the state:

$$\begin{aligned} S' \;&= \mathtt{A}[0:\mathtt{T}_1, 6:\mathtt{T}_2, 0:\mathtt{ETH}] \mid \mathtt{B}[0:\mathtt{ETH}] \\ &\mid \mathtt{PriceOracle}[1:\mathtt{ETH}, \mathtt{owner}=\mathtt{B}, \mathtt{fee}=1, \mathtt{prices}=\{\mathtt{T}_1=3, \mathtt{T}_2=5\}, \ldots] \end{aligned}$$

[1] The ERC-20 interface uses different names for token operations, i.e. `approve` to authorize token transfers, and `transferFrom` to receive previously authorized tokens.

$$\mid \texttt{Exchange}[10 : \texttt{T}_1, 94 : \texttt{T}_2, \texttt{p} = \texttt{PriceOracle}, \texttt{t0} = \texttt{T}_1, \texttt{t1} = \texttt{T}_2]$$

Note that firing X in S' would revert, since the sender A does not have enough units of $\texttt{T}_1$ (and of ETH) to transfer to the Exchange. ⋄

Sender-agnostic methods. We say that a contract method is *sender-agnostic* when the effect of an internal call to the method does not depend on the identity of the caller. More precisely, consider two transactions $\texttt{X}_0$ and $\texttt{X}_1$ with the same initiator, whose executions trigger internal calls to the method f of contract C:

$$\texttt{D}_0 : \texttt{C.f(args)} \qquad \texttt{D}_1 : \texttt{C.f(args)}$$

in the same state, with the same arguments, but with different senders $\texttt{D}_0$ and $\texttt{D}_1$, both of which have approved the token transfers needed to fund the respective calls. We then say that f is sender-agnostic whenever the two internal calls either both revert, or both succeed producing the same effects, up-to the transfer of tokens to the sender. More specifically, the two calls update the contract state in the same way, transfer the same tokens to the same user accounts, perform the same internal calls to other methods, and transfer the same tokens to their respective senders $\texttt{D}_0$ and $\texttt{D}_1$.

Example 2. In the PriceOracle contract in Listing 1, the method setPrice is *not* sender-agnostic whenever owner is a contract account with a method that can call setPrice. In that case, the effect of an internal call to setPrice depends on the sender: if it is the owner, the call succeeds, otherwise it reverts. Instead, the method getRate is sender-agnostic, as its effect does not depend on the identity of the caller. For the same reason, the method swap of the Exchange contract in Listing 2 is sender-agnostic as well. ⋄

Token flows. Let S be a blockchain state and let C, D be sets of contract accounts. We say that there is a *token flow* from C to D in S if there exists a sequence of valid transactions in S containing as effects an output of some token type T from C and an input of T from D. As a special case, we say that there is a *direct token flow* from C to D in S if there exists a sequence of valid transactions from S where the effects of one of the transactions includes the transfer of tokens from some contract in C to some contract in D.

```
contract Airdrop {
  token t;
  constructor(token t_, uint n) { t=t_; t.receive(n); }
  function drop(uint n) public { t.send(msg.sender,n); }
}

contract FreeExchange is IExchange {
  IExchange ex;

  constructor(IExchange ex_) { ex=ex_; }

  function getTokens() public view returns (token[] memory) { return ex.getTokens(); }

  function getRate(token ti, token to) public view returns (uint) {
    return ex.getRate(ti,to);
  }

  function swap(token ti, uint x, token to, uint ymin) public returns (uint) {
    ti.receive(x);
    uint fee = PriceOracle.fee();
    Airdrop.drop(fee);
    ti.approve(ex,x);
    ETH.approve(ex,fee);
    ex.swap(ti,x,to,ymin); // revert if pair (ti,to) not swappable
    uint y = T2.balanceOf(this);
    require (y>=ymin);
    to.send(msg.sender,y);
    return y;
  }
}
```

Listing 3. An airdrop contract and a fee-less exchange contract.

Example 3. Let $C = \{\texttt{Exchange}\}$ and let $D = \{\texttt{PriceOracle}\}$ for the contracts of Listings 1 and 2, and let S as in Example 2. As shown before, firing $\texttt{A} : \texttt{Exchange.swap}(\texttt{T}_1, 10, \texttt{T}_2, 0)$ in S triggers an internal call:

$$\texttt{Exchange} : \texttt{PriceOracle.getRate}(\texttt{T}_1, \texttt{T}_2)$$

whose effects include a transfer of ETH from the caller Exchange to the callee PriceOracle. Hence, there is a *direct* token flow from C to D in S.

To illustrate *in*direct token flows, consider the contracts in Listing 3. The contract Airdrop allows anyone to withdraw any amount of a given token deposited in the contract. This feature is leveraged by the contract FreeExchange to enable fee-less token swaps. To do that, the FreeExchange first withdraws the needed fee from the Airdrop, then invokes the Exchange from Listing 2 to perform the token swap, and finally transfers back the tokens to the sender.

Let $C = \{\texttt{Airdrop}\}$, let $D = \{\texttt{PriceOracle}\}$, and let

$$S_1 = S \mid \texttt{Airdrop}[1000 : \texttt{ETH}, \texttt{t} = \texttt{ETH}] \mid \texttt{FreeExchange}[\texttt{ex} = \texttt{Exchange}]$$

The transaction $\texttt{A} : \texttt{FreeExchange.swap}(10)$ causes an ETH flow from C to D in S_1. This flow is indirect, since none of the internal calls includes a transfer of ETH from C to D as an effect. However, the effect of the internal call to Airdrop is an output of ETH from C (to the FreeExchange), and the effect of the internal call to PriceOracle includes an input of ETH (from Exchange) to D. ◇

3. Maximal extractable value

In this section we introduce a notion of MEV tailored to the study of smart contract compositions. In Section 3.1, we formally define MEV and compare our notion with the seminal definition introduced in [8]. In Section 3.2 we establish a collection of basic properties of MEV that are used throughout the paper. In Sections 3.3 and 3.4, we prove two key results that provide sufficient conditions under which the adversary's capabilities can be restricted while preserving MEV.

3.1. Definition

We measure the effect of an attack as the variation in the *wealth* of a set of victim accounts before and after the attack. Given a blockchain state S and set of accounts $\mathcal{A}$ (possibly comprising both user and contract accounts), we define their wealth $\$_{\mathcal{A}}(S)$ as the weighted sum of the tokens held in $\mathcal{A}$'s wallets, where the weights are the token prices. We denote by $\$\mathbf{1}_{\texttt{T}}$ the *price* of a token type T and assume that $\$\mathbf{1}_{\texttt{T}} > 0$ for all T. We assume these prices to be fixed throughout an attack (note instead that the prices given by a PriceOracle contracts can vary, since they depend on the blockchain state).

Definition 1 (Wealth)**.** The wealth of a set of accounts $\mathcal{A} \subseteq \mathbb{A}$ in S is given by:

$$\$_{\mathcal{A}}(S) \;=\; \sum_{\texttt{T} \in \mathbb{T}} \omega_{\mathcal{A}}(S)(\texttt{T}) \cdot \$\mathbf{1}_{\texttt{T}}$$

Note that since both wallets and blockchain states have finite domain, states cannot have an infinite amount of tokens, and so the wealth is always finite.

Definition 2 (Gain)**.** The gain of $\mathcal{A} \subseteq \mathbb{A}$ upon firing $\vec{\mathcal{X}}$ in S is given by:

$$\gamma_{\mathcal{A}}(S, \vec{\mathcal{X}}) \;=\; \$_{\mathcal{A}}(S') - \$_{\mathcal{A}}(S) \qquad \text{if } S \xrightarrow{\vec{\mathcal{X}}} S'$$

MEV was originally defined in [8] as the maximum gain that an adversary can achieve by firing a sequence of transactions in a blockchain state S. If we denote by $\kappa(\mathcal{M})$ the set of transactions craftable by $\mathcal{M}$ by using their private knowledge, and by $\kappa(\mathcal{M})^*$ their finite sequences, the notion of MEV in [8] can be translitterated into our notation as:

$$\text{MEV}(S) = \max \left\{ \gamma_{\mathcal{M}}(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa(\mathcal{M})^* \right\} \tag{2}$$

While (2) measures an adversary's success in terms of its *gain*, to analyse the economic security for smart contract compositions we focus instead on the *loss* an adversary can inflict, on a given set of victim contracts C, by manipulating transaction ordering and exploiting cross-contract dependencies. To this purpose, our notion of MEV distinguishes from (2) in the following aspects:

1. We measure the value that adversaries can *drain* from a given set of victim contracts C. This means that only the tokens extracted from the contracts in C contribute to the MEV. Unlike in (2), the tokens grabbed from other contracts do not contribute to MEV.
2. We count as MEV *all* the tokens that $\mathcal{M}$ can remove from C, regardless of whether $\mathcal{M}$ can transfer them to their wallets. Namely, while (2) maximises $\mathcal{M}$'s gain $\gamma_{\mathcal{M}}$, our notion maximises C's *loss* $-\gamma_C$. Doing so, our notion of secure composability also encompasses *irrational adversaries* who try to damage the victim contracts without necessarily making a profit.
3. We parameterise our MEV w.r.t. the set D of contracts *callable* by $\mathcal{M}$. This enables us to reinterpret the separation between private and public information in language-based non-interference. In that setting, the main insight is that observable (public) outputs remain unaffected by variations in confidential (private) inputs — equivalently, restricting inputs to the private ones must preserve the public outputs. In our context, we rephrase this by requiring that the loss inflictable to C (the public output) is not affected when restricting the set of callable contracts to a subset D (the private inputs).

Definition 3 (MEV). Let $\kappa_D(\mathcal{M}) = \{\mathsf{X} \in \kappa(\mathcal{M}) \mid callee(\mathsf{X}) \in D\}$ be the set of transactions craftable by $\mathcal{M}$ restricted to calling contracts in D. We define:

$$\mathrm{MEV}_D(S, C) = \max\left\{-\gamma_C(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^*\right\} \tag{3}$$

Hereafter, we refer to the sets C and D as the *victim* and *callable* contracts, respectively, and we abbreviate $\mathrm{MEV}_{\mathbb{A}_c}(S, C)$ as $\mathrm{MEV}(S, C)$.

Although the above notion of MEV is meaningful for arbitrary choices of the set of victims, when applying this notion to define MEV non-interference we will always choose the victims to be the newly deployed contracts (see Definition 8). The intuition is that MEV non-interference detects whether deploying a new contract will allow an adversary to inflict some losses to it by leveraging its dependencies.

Example 4. Consider the `Airdrop` contract from Listing 3, and let:

$$S = S' \mid \texttt{Airdrop}[n : \mathtt{T}, \mathtt{t} = \mathtt{T}]$$

where S' is arbitrary. Let $C = \{\texttt{Airdrop}\}$, and let D be any set of contract accounts including C. Any user can fire a transaction in S to withdraw the entire airdrop balance. Therefore, $\mathrm{MEV}_D(S, C) = n \cdot \$\mathbf{1}_{\mathtt{T}}$. ◇

Lemma 1 relates our notion of MEV with that in (2). In particular, when the set of victim contracts is the universe $\mathbb{A}_c$, our MEV over-approximates (2). Note that the inequality $\leq$ does not hold in general, since some of the tokens drained from contracts in $\mathbb{A}_c$ could be transferred to some account not in $\mathcal{M}$, and therefore they would contribute to the loss of $\mathbb{A}_c$ but not to the gain of $\mathcal{M}$.

Proposition 1. $\mathrm{MEV}_{\mathbb{A}_c}(S, \mathbb{A}_c) \geq \mathrm{MEV}(S)$.

3.2. Basic properties of MEV

Lemma 2 establishes some basic properties of MEV:

- Item 1 studies some border cases when the MEV is zero.
- Item 2 states that the MEV extractable from C is non-negative and bounded by the wealth of C.
- Item 3 states that the MEV of S is preserved when restricting the set of callable/victim contracts to the set $\dagger S$ of contracts occurring in S.
- Items 4 and 5 state that widening the callable contracts or the contract state may potentially increase MEV. Example 5 shows, instead, that widening the set of victim contracts does not necessarily increase the MEV: this is because maximising the loss of a contract may increase the gain of another one.

Lemma 2. *For all S, C, D, D' and Δ:*

1. $\mathrm{MEV}_D(S, \emptyset) = \mathrm{MEV}_\emptyset(S, C) = 0$
2. $0 \leq \mathrm{MEV}_D(S, C) \leq \$_C(S)$
3. $\mathrm{MEV}_D(S, C) = \mathrm{MEV}_D(S, C \cap \dagger S) = \mathrm{MEV}_{D \cap \dagger S}(S, C)$
4. *if $D \subseteq D'$, then* $\mathrm{MEV}_D(S, C) \leq \mathrm{MEV}_{D'}(S, C)$
5. $\mathrm{MEV}_D(S, C) \leq \mathrm{MEV}_D(S \mid \Delta, C)$

Additionally, items 1–4 also hold for non well-formed states S.

Example 5. We show that $C \subseteq C'$ does not imply $\mathrm{MEV}_D(S, C) \leq \mathrm{MEV}_D(S, C')$. Consider the following contracts, and assume that tokens have unit prices:

```
contract C0 { function f() { T0.send(msg.sender,5); } }
contract C1 { function f() { T0.receive(5); T1.send(msg.sender,1); } }
contract C2 { function f() { T1.receive(1); T2.send(msg.sender,100); } }
```

Let $\mathcal{M} = \{\mathtt{M}\}$, and let $\vec{\mathcal{X}} = \mathtt{M} : \mathtt{C0.f()}\ \mathtt{M} : \mathtt{C1.f()}\ \mathtt{M} : \mathtt{C2.f()}$. We have that:

$$S = \mathtt{M}[0 : \mathtt{T}_2] \mid \mathtt{C0}[5 : \mathtt{T}_0] \mid \mathtt{C1}[1 : \mathtt{T}_1] \mid \mathtt{C2}[100 : \mathtt{T}_2] \xrightarrow{\vec{\mathcal{X}}} \mathtt{M}[100 : \mathtt{T}_2] \mid \mathtt{C0}[0 : \mathtt{T}_0] \mid \mathtt{C1}[5 : \mathtt{T}_0] \mid \mathtt{C2}[1 : \mathtt{T}_1]$$

Let $C = \{\mathtt{C2}\} \subseteq \{\mathtt{C1}, \mathtt{C2}\} = C'$. Note that $\vec{\mathcal{X}}$ maximizes the loss of both C and C'. Indeed, `C2` is completely drained, except for the $1 : \mathtt{T}_1$ that is required to call its `f`. For C', the only alternative action would be to call `C1.f` without calling `C2.f`, but this would produce a gain instead of a loss. Therefore:

$$\mathrm{MEV}(S, \{\mathtt{C2}\}) = -\gamma_{\{\mathtt{C2}\}}(S, \vec{\mathcal{X}}) = 100 - 1 = 99 \not\leq \mathrm{MEV}(S, \{\mathtt{C1}, \mathtt{C2}\}) = -\gamma_{\{\mathtt{C1}, \mathtt{C2}\}}(S, \vec{\mathcal{X}}) = 101 - 6 = 95$$ ◇

Example 6. Lemma 2(5) ceases to hold if $S \mid \Delta$ is not well-formed and contracts are allowed to test the existence of other contracts. In such a setting, adding contracts to the system may strictly reduce MEV, violating monotonicity. E.g., consider a contract `C` exposing a method `f` with the following behaviour: `f` performs a call to another contract `D`; if the call fails (e.g. because `D` does not exist), then `f` transfers `C`'s `T` balance to the caller; otherwise, it reverts. Let S be a state containing `C`$[1 : \mathtt{T}]$ but not `D`, and let Δ be a state containing `D`. If $C = D = \{\mathtt{C}\}$, then $\mathrm{MEV}_D(S, C) = 1 \cdot \$\mathbf{1}_{\mathtt{T}} > 0$, while $\mathrm{MEV}_D(S \mid \Delta, C) = 0$. ◇

Lemma 3 allows us to narrow the blockchain state while preserving MEV. Specifically, if D is the set of callable contracts, we can remove from the state all the contracts that are neither in D nor in its dependencies. Recall that the states mentioned in our statements are implicitly assumed to be well-formed: in particular, in Lemma 3 the well-formedness of $S \mid \Delta$ implies that the contracts in S cannot call those in Δ.

Lemma 3 (State narrowing). $\mathrm{MEV}_D(S \mid \Delta, C) = \mathrm{MEV}_D(S, C)$ *if* $D \subseteq \dagger S$.

The following example shows that if the condition $D \subseteq \dagger S$ in Lemma 3 does not hold, then MEV preservation is not guaranteed.

Example 7. Consider the following state containing instances of the `PriceOracle` and `Exchange` contracts in Listing 2, which overall allow anyone to swap $1 : \mathtt{T}_1$ for $2 : \mathtt{T}_2$:

$$\begin{aligned} S \;=\; & \mathtt{M}[1 : \mathtt{T}_2] \mid \mathtt{PriceOracle1}[\mathtt{fee} = 0, \mathtt{prices} = \{\mathtt{T}_1 = 2, \mathtt{T}_2 = 1\}, \ldots] \\ & \mid \mathtt{Exchange1}[0 : \mathtt{T}_1, 10 : \mathtt{T}_2, \mathtt{p} = \mathtt{PriceOracle1}, \mathtt{t0} = \mathtt{T}_1, \mathtt{t1} = \mathtt{T}_2] \end{aligned}$$

Assume that the tokens are equally priced, e.g. $\$\mathbf{1}_{\mathtt{T}_1} = \$\mathbf{1}_{\mathtt{T}_2} = 1$. The adversary `M` cannot inflict any loss to `Exchange1`: indeed, `M` only has $1 : \mathtt{T}_2$, and so the only exchange she could attempt is to sell $\mathtt{T}_2$ to buy $\mathtt{T}_1$. However, `Exchange1` has not units of $\mathtt{T}_1$, so no interaction with `M` is possible.

Assume now to extend S with the following Δ:

$$\begin{aligned} \Delta \;=\; & \mathtt{PriceOracle2}[\mathtt{fee} = 0, \mathtt{prices} = \{\mathtt{T}_1 = 1, \mathtt{T}_2 = 1\}, \ldots] \\ & \mid \mathtt{Exchange2}[1 : \mathtt{T}_1, 0 : \mathtt{T}_2, \mathtt{p} = \mathtt{PriceOracle2}, \mathtt{t0} = \mathtt{T}_2, \mathtt{t1} = \mathtt{T}_1] \end{aligned}$$

which allows anyone to swap tokens $\mathtt{T}_2$ with an equal amount of tokens $\mathtt{T}_1$. In particular, `M` can use `Exchange2` to obtain $1 : \mathtt{T}_1$ for $1 : \mathtt{T}_2$, and then exchange it for $2 : \mathtt{T}_2$ through `Exchange1`. Let $C = \{\mathtt{Exchange1}\}$ and let $D = \mathbb{A}_c$. Since $D \not\subseteq \dagger S$, Lemma 3 does not apply and so MEV preservation is not guaranteed. Indeed:

$$\mathrm{MEV}_D(S, \{\mathtt{Exchange1}\}) = 0 \qquad \neq \qquad \mathrm{MEV}_D(S \mid \Delta, \{\mathtt{Exchange1}\}) = 2 \cdot \$\mathbf{1}_{\mathtt{T}_2} - 1 \cdot \$\mathbf{1}_{\mathtt{T}_1} = 1 \qquad \diamond$$

3.3. Front-running resistance

Consider a state S where a user `A` is deploying new contracts Δ, with the expectation that the loss inflictable to Δ is bounded by $\mathrm{MEV}(S \mid \Delta, \dagger\Delta)$. This expectation is generally false, since an adversary could front-run `A`'s deployment transaction with the deployment of new contracts Γ, and in this way increase the loss inflictable to Δ. Theorem 5 gives sufficient conditions under which MEV is preserved by such front-running attacks. This result will be the basis to establish front-running resistance for MEV non-interference.

The proof of Theorem 5 leverages a fundamental MEV preservation result, established by Lemma 4:

$$\mathrm{MEV}_D(S, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(S, C) \qquad \text{(under given conditions)}$$

The key intuition for proving this result is to take a sequence of transactions $\vec{\mathcal{X}}$ that maximizes $\mathrm{MEV}_D(S, C)$, and to translate it into a sequence $\vec{\mathcal{Y}}$ where:

- we preserve the transactions with callee in $deps(C)$, and
- replace those with callee in $D \setminus deps(C)$ with transactions where the adversary directly performs the first internal call in the *boundary* between C and D, i.e. the contracts in $deps(C)$ that are called from some contract in $deps(D) \setminus deps(C)$ (see Definition 4).

In order for $\vec{\mathcal{Y}}$ to be valid and have the same gain as $\vec{\mathcal{X}}$, a few conditions are sufficient:

- the contracts in the boundary are sender-agnostic, i.e. they are not aware of the identity of the caller. This guarantees that replacing the internal call with the external call (with sender in $\mathcal{M}$) does not affect the semantics of the called functions;
- the adversary has enough tokens to fund the boundary transactions. Technically, we ensure this by requiring that there are no token flows between the dependencies of D and the boundary.

Definition 4 (Boundary). We define the *boundary* between C and D as:

$$\partial(C, D) \;=\; \left\{ \mathtt{C} \in deps(C) \,\middle|\, \exists \mathtt{D} \in deps(D) \setminus deps(C) \;:\; \mathtt{D} \text{ calls } \mathtt{C} \right\}$$

Lemma 4 (Callable contracts narrowing). *The equality:*

$$\mathrm{MEV}_D(S, C) = \mathrm{MEV}_{D \cap deps(C)}(S, C)$$

holds if the following conditions are satisfied:

1. *the contracts in* $\partial(C, D)$ *are sender-agnostic;*
2. $\partial(C, D) \subseteq D$*;*
3. *there is no token flow from* $D \setminus deps(C)$ *to* $D \cap deps(C)$ *in* S*;*
4. *there is no direct token flow from* $deps(D) \setminus deps(C)$ *to* $\partial(C, D)$ *in* S*.*

Example 8 illustrates our proof by applying its construction to a toy example. Technical Examples 20 to 22 show that, when the above conditions are not met, then MEV preservation is not guaranteed to hold.

Example 8. To illustrate Lemma 4, consider the contracts:

```
contract C0 { function f() { T.receive(1); T.send(M,2); } }
contract C1 { function f() { T.send(C0,1); C0.f(); } }
contract C2 { function f() { require (msg.sender==C3); C1.f(); } }
contract C3 { function f() { C2.f(); C1.f(); } }
```

Let $\mathcal{M} = \{\mathtt{M}\}$, $\mathcal{D} = \{\mathtt{C1}, \mathtt{C3}\}$, $\mathcal{C} = \{\mathtt{C0}, \mathtt{C1}\}$, and let:

$$S = \mathtt{M}[0:\mathtt{T}] \mid \mathtt{C0}[2:\mathtt{T}] \mid \mathtt{C1}[2:\mathtt{T}] \mid \mathtt{C2}[0:\mathtt{T}] \mid \mathtt{C3}[0:\mathtt{T}]$$

Let $\vec{\mathscr{X}} = \mathtt{M} : \mathtt{C3.f}() \in \kappa_{\mathcal{D}}(\mathtt{M})^*$. By executing $\vec{\mathscr{X}}$ in S, we have that:

$$S \xrightarrow{\mathtt{M}:\mathtt{C3.f}()} \mathtt{M}[4:\mathtt{T}] \mid \mathtt{C0}[0:\mathtt{T}] \mid \mathtt{C1}[0:\mathtt{T}] \mid \mathtt{C2}[0:\mathtt{T}] \mid \mathtt{C3}[0:\mathtt{T}]$$

Since there are no tokens left in $\mathcal{C}$, $\vec{\mathscr{X}}$ clearly maximises the loss of $\mathcal{C}$, hence:

$$\mathrm{MEV}_{\mathcal{D}}(S, \mathcal{C}) = 4 \cdot \$\mathbf{1}_{\mathtt{T}}$$

We first check that the conditions of Lemma 4 are satisfied. We have that:

$$\begin{aligned}\partial(\mathcal{C}, \mathcal{D}) &= \left\{\mathtt{C} \in \mathit{deps}(\mathcal{C}) \,\middle|\, \exists \mathtt{D} \in \mathit{deps}(\mathcal{D}) \setminus \mathit{deps}(\mathcal{C}) \;:\; \mathtt{D} \text{ calls } \mathtt{C}\right\} \\ &= \left\{\mathtt{C} \in \{\mathtt{C0}, \mathtt{C1}\} \,\middle|\, \exists \mathtt{D} \in \{\mathtt{C2}, \mathtt{C3}\} \;:\; \mathtt{D} \text{ calls } \mathtt{C}\right\} \\ &= \{\mathtt{C1}\}\end{aligned}$$

The conditions of Lemma 4 are then satisfied, since:

(1) the contract $\mathtt{C1} \in \partial(\mathcal{C}, \mathcal{D})$ is sender-agnostic. Note that C2 is not sender-agnostic: however, this does not violate assumption 1, since only the contracts in $\partial(\mathcal{C}, \mathcal{D})$ are required to be sender-agnostic.
(2) $\partial(\mathcal{C}, \mathcal{D}) = \{\mathtt{C1}\} \subseteq \mathcal{D}$.
(3) There is no token flow from $\mathcal{D} \setminus \mathit{deps}(\mathcal{C}) = \{\mathtt{C3}\}$ to $\mathcal{D} \cap \mathit{deps}(\mathcal{C}) = \{\mathtt{C1}\}$, since C3 has no token output.
(4) There is no direct token flow from $\mathit{deps}(\mathcal{D}) \setminus \mathit{deps}(\mathcal{C}) = \{\mathtt{C2}, \mathtt{C3}\}$ to $\partial(\mathcal{C}, \mathcal{D}) = \{\mathtt{C1}\}$ in S, since there is no token transfer towards C1.

We now construct a sequence of transactions $\vec{\mathscr{Y}} \in \kappa_{\mathcal{D} \cap \mathit{deps}(\mathcal{C})}(\mathtt{M})^* = \kappa_{\{\mathtt{C1}\}}(\mathtt{M})^*$ that inflicts on $\mathcal{C}$ the same loss as $\vec{\mathscr{X}}$, following the proof of Lemma 4. Let $\vec{x}$ be the sequence of calls triggered by $\vec{\mathscr{X}}$. We obtain a sequence $\vec{\mathscr{Y}}$ from $\vec{x}$ as follows:

1. external calls $\mathtt{M} : \mathtt{C.f}(\mathtt{args})$ in $\vec{x}$ where $\mathtt{C} \in \mathit{deps}(\mathcal{C})$ are preserved in $\vec{\mathscr{Y}}$;
2. internal calls $\mathtt{C} : \mathtt{D.f}(\mathtt{args})$ in $\vec{x}$ where $\mathtt{C} \in \partial(\mathcal{C}, \mathcal{D})$ — i.e., $\mathtt{C} \notin \mathit{deps}(\mathcal{C})$ and $\mathtt{D} \in \mathit{deps}(\mathcal{C})$ — are added to $\vec{\mathscr{Y}}$, replacing the sender C with the originator M;
3. all the other calls in $\vec{x}$ are discarded.

The sequence $\vec{x}$ of (external/internal) calls induced by $\vec{\mathscr{X}}$ and the constructed sequence of transactions $\vec{\mathscr{Y}}$ are the following:

$\vec{x} =$	M : C3.f()	C3 : C2.f()	C2 : C1.f()	C1 : C0.f()	C3 : C1.f()	C1 : C0.f()
$\vec{\mathscr{Y}} =$			M : C1.f()		M : C1.f()	

Note that $\vec{\mathscr{Y}} \in \kappa_{\mathcal{D} \cap \mathit{deps}(\mathcal{C})}(\mathtt{M})^* = \kappa_{\{\mathtt{C1}\}}(\mathtt{M})^*$. By executing $\vec{\mathscr{Y}}$ in S, we obtain:

$$\begin{aligned}S &\xrightarrow{\mathtt{M}:\mathtt{C1.f}()} \mathtt{M}[2:\mathtt{T}] \mid \mathtt{C0}[1:\mathtt{T}] \mid \mathtt{C1}[1:\mathtt{T}] \mid \mathtt{C2}[0:\mathtt{T}] \mid \mathtt{C3}[0:\mathtt{T}] \\ &\xrightarrow{\mathtt{M}:\mathtt{C1.f}()} \mathtt{M}[4:\mathtt{T}] \mid \mathtt{C0}[0:\mathtt{T}] \mid \mathtt{C1}[0:\mathtt{T}] \mid \mathtt{C2}[0:\mathtt{T}] \mid \mathtt{C3}[0:\mathtt{T}]\end{aligned}$$

Hence, $\mathrm{MEV}_{\mathcal{D} \cap \mathit{deps}(\mathcal{C})}(S, \mathcal{C}) = 4 \cdot \$\mathbf{1}_{\mathtt{T}}$, confirming the preservation of inflictable loss stated by Lemma 4. ◇

Theorem 5 (MEV and front-running resistance). *Let $\mathcal{C} = \dagger\Delta$. Then:*

$$\mathrm{MEV}_{\mathcal{D}}(S \mid \Gamma \mid \Delta, \mathcal{C}) \;=\; \mathrm{MEV}_{\mathcal{D} \cap \mathit{deps}(\mathcal{C})}(S \mid \Delta, \mathcal{C})$$

holds if the following conditions are satisfied:

1. *the contracts in $\mathit{deps}(\Delta)$ are sender-agnostic*
2. *$\mathit{deps}(\Delta) \subseteq \mathcal{D}$*
3. *there are no token flows from $\mathit{deps}(\mathcal{D}) \setminus \mathit{deps}(\Delta)$ to $\mathit{deps}(\Delta)$ in $S \mid \Gamma \mid \Delta$.*

Proof. The conditions of Lemma 4 are satisfied:

(1) $\partial(C, D)$ are sender-agnostic, since $\partial(C, D) \subseteq deps(C) = deps(\Delta)$ and, by hypothesis 1, the contracts in $deps(\Delta)$ are sender-agnostic;
(2) $\partial(C, D) \subseteq deps(\Delta) \subseteq D$ by hypothesis 2;
(3) No token flows from $D \setminus deps(C)$ to $D \cap deps(C)$ in $S \mid \Gamma \mid \Delta$. Indeed, we have $D \setminus deps(C) \subseteq deps(D) \setminus deps(\Delta)$ and $D \cap deps(C) \subseteq deps(\Delta)$, hence we conclude by hypothesis 3.
(4) No direct token flows from $deps(D) \setminus deps(C)$ to $\partial(C, D)$. Indeed, we have $deps(D) \setminus deps(C) = deps(D) \setminus deps(\Delta)$ and $\partial(C, D) \subseteq deps(\Delta)$, hence we conclude by hypothesis 3.

Therefore, by Lemma 4 it follows that:

$$\mathrm{MEV}_D(S \mid \Gamma \mid \Delta, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(S \mid \Gamma \mid \Delta, C)$$

Now, since $S \mid \Gamma$ and $S \mid \Delta$ are well-formed, then $deps(\Gamma)$ is disjoint from $\dagger\Delta$ and $deps(\Delta)$ is disjoint from $\dagger\Gamma$, and so the states $S \mid \Gamma \mid \Delta$ and $S \mid \Delta \mid \Gamma$ are equivalent. Then:

$$\mathrm{MEV}_{D \cap deps(C)}(S \mid \Gamma \mid \Delta, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(S \mid \Delta \mid \Gamma, C)$$

Since $S \mid \Delta$ is well-formed, then $deps(\Delta) \subseteq \dagger(S \mid \Delta)$. Then, since $D \cap deps(C) \subseteq deps(\Delta) \subseteq \dagger(S \mid \Delta)$, by Lemma 3 we have:

$$\mathrm{MEV}_{D \cap deps(C)}(S \mid \Delta \mid \Gamma, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(S \mid \Delta, C)$$

Summing up, we obtain the thesis:

$$\mathrm{MEV}_D(S \mid \Gamma \mid \Delta, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(S \mid \Delta, C)$$ □

3.4. Observational invariance

We now provide another MEV preservation result, which will be the basis to establish sufficient conditions for MEV non-interference in Section 5.

Theorem 6 states that the MEV inflictable to Δ in $S \mid \Delta$ is preserved by narrowing the callable contracts to Δ when Δ is *observationally invariant* w.r.t. adversarial moves in S. Roughly, this means that any adversarial transaction calling a contract in S cannot affect what Δ observes by invoking the functions in S. The observables include the returned values, the transferred tokens and the aborts of functions that can be called from Δ. In other words, even if the adversary interacts with S, possibly changing its internal state, the contracts in Δ will have the same behaviour as if the adversary never interacted with S.

Definition 5 (Observational invariance)**.** Let $S \mid \Delta$ be a blockchain state (not necessarily well-formed). We say that Δ is *observationally invariant in* S whenever the following condition holds. Let S' be any state reachable through moves of $\mathcal{M}$ from $S \mid \Delta$. Then, we require that:

1. if $\mathtt{X} \in \kappa(\mathcal{M})$ is such that $callee(\mathtt{X}) \in \dagger S$, then executing $\mathtt{X}$ in S' does not change the state of the contracts in Δ (neither their wallet nor storage);
2. if $\mathtt{X}, \mathtt{Y} \in \kappa(\mathcal{M})$ are such that $callee(\mathtt{X}) \in \dagger S$, $callee(\mathtt{Y}) \in \dagger\Delta$, and the sequence $\mathtt{XY}$ is valid in S', then $\mathtt{Y}$ satisfies the following conditions:
   - $\mathtt{Y}$ is valid in S' provided that the sender has enough tokens;
   - executing $\mathtt{Y}$ in S' triggers the same sequence of internal calls from the contracts in $\dagger\Delta$ to those in $\dagger S$, with the same arguments and sent tokens, as the sequence of calls that $\mathtt{XY}$ triggers in S';
   - each call triggered by $\mathtt{Y}$ has the same effects (of the kinds: aborts, token transfers, method calls to $\dagger\Delta$, returns) of the corresponding call of $\mathtt{XY}$.

Example 9. Let $\Delta = \mathtt{Exchange}[\cdots]$, where the exchange uses an instance of the `PriceOracle` contract occurring in S. Let $\mathcal{M}$ be a set of accounts not including the oracle's owner. Then, Δ is observationally invariant w.r.t. $\mathcal{M}$'s moves in S. To show that, let S' be any state reachable from $S \mid \Delta$ through adversarial moves, and let $\mathtt{X}$ and $\mathtt{Y}$ satisfy the hypotheses of Definition 5. The transaction $\mathtt{X}$ must call `PriceOracle`; since $\mathtt{X}$ is valid, it must call the method `getRate`, since `setPrice` requires that the caller is the owner. Since $\mathtt{Y}$ is valid after $\mathtt{X}$ is executed and $\mathtt{X}$ does not modify the `PriceOracle`'s storage, then $\mathtt{Y}$ is also valid in S', and it produces exactly the same effects when executed in both states. ◇

Example 10. Let S consist of the `Bank` contract in Listing 4. We show that any Δ is observationally invariant w.r.t. adversarial moves in S. Let S' be any state reachable from $S \mid \Delta$ through adversarial moves, and let $\mathtt{X}$, $\mathtt{Y}$ satisfy the hypotheses of Definition 5. Then, $\mathtt{X}$ must call one of the methods in `Bank`. The effects of both methods `deposit` and `withdraw` is to update the `credits` entry of the adversary and its token balance. By assumption, $\mathtt{Y}$ calls a method of a contract `D` in Δ, and of course $\mathtt{D} \notin \mathcal{M}$. The calls from `D` to `Bank` are not affected by $\mathtt{X}$, since they depend on the `credits` of `D`. Then, since $\mathtt{Y}$ is valid after firing $\mathtt{X}$, it is also valid in S' — provided that the $\mathtt{Y}$'s sender has enough tokens to fund the call. Finally, the effects of $\mathtt{Y}$ are not affected by the execution of $\mathtt{X}$. ◇

Theorem 6 (MEV and observational invariance)**.** *For any state $S \mid \Delta$ (not necessarily well-formed),*

$$\mathrm{MEV}_D(S \mid \Delta, \dagger\Delta) \;=\; \mathrm{MEV}_{D \cap \dagger\Delta}(S \mid \Delta, \dagger\Delta)$$

```
contract Bank {
  mapping (address user => uint credit) credits;

  // receive tokens T from the sender and store them in the contract
  // (inreasing the sender's credits of the corresponding amount)
  function deposit(uint amount) public {
    T.receive(amount);
    credits[msg.sender] += amount;
  }

  // transfers tokens T from the contract to the sender
  // (decreasing the sender's credits of the corresponding amount)
  function withdraw(uint amount) public {
    require(amount <= credits[msg.sender]);
    credits[msg.sender] -= amount;
    T.send(msg.sender,amount);
  }
}
```

Listing 4. A bank contract.

holds whenever both the following conditions are satisfied:

1. *there are no token flows from S to Δ in $S \mid \Delta$;*
2. *Δ is observationally invariant w.r.t. adversarial moves in S.*

Proof. The inequality $\geq$ is given by Lemma 2(4), so we prove $\leq$. Let $\vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^*$ be a sequence of transactions that maximizes the loss of $\dagger\Delta$ when executed $S \mid \Delta$. W.l.o.g., assume that such transactions are valid. We must show that there is a sequence $\vec{\mathcal{Y}} \in \kappa_{D\cap\dagger\Delta}(\mathcal{M})^*$ that causes a loss of $\dagger\Delta$ greater than or equal to the one caused by $\vec{\mathcal{X}}$. We choose $\vec{\mathcal{Y}}$ to be the subsequence of $\vec{\mathcal{X}}$ comprising only direct calls to contracts in Δ. Clearly, $\vec{\mathcal{Y}}$ is in $\kappa_{D\cap\dagger\Delta}(\mathcal{M})^*$.

We now show that $\vec{\mathcal{Y}}$ is valid in $S \mid \Delta$. First, we note that any transaction of $\vec{\mathcal{X}}$ calling contracts in S (i.e. the ones that have been removed to form $\vec{\mathcal{Y}}$) does not change the behaviour of S as observed from Δ. This is due to the observational invariance of Δ (Definition 5): the removed transactions can affect the state of S but this does not alter the effects of the contract calls from Δ to S. Consequently, Δ can not distinguish between the state of S reached by $\vec{\mathcal{X}}$ and the one reached by $\vec{\mathcal{Y}}$, hence Δ will exhibit the same behavior in both cases.

For this reason, $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$ both cause the same state updates in Δ, and in particular they perform the same token transfers to and from it. To conclude that $\vec{\mathcal{Y}}$ is valid, we note that the adversary must have enough tokens to fund all the calls in $\vec{\mathcal{Y}}$. Indeed, the adversary has enough funds to execute $\vec{\mathcal{X}}$, and any token that they gain from the discarded transactions cannot be used by calls in $\vec{\mathcal{Y}}$, since there is no token flow from S to Δ.

Since $\vec{\mathcal{Y}}$ is valid and updates the state of Δ in the same way as $\vec{\mathcal{X}}$, the loss caused to Δ must also be the same, so $\mathrm{MEV}_D(S \mid \Delta, \dagger\Delta) \leq \mathrm{MEV}_{D\cap\dagger\Delta}(S \mid \Delta, \dagger\Delta)$, which concludes the proof. □

4. MEV for wealthy adversaries

We now study MEV under a stronger adversarial model, in which the adversary is assumed to always possess enough tokens to inflict the maximal possible loss on the victim contracts. This analysis relies a basic property of the underlying blockchain model, which we call *wallet monotonicity*: any transaction that is valid in S must produce the same effect if we enrich the users' wallets in S with additional tokens.

We formalise wallet monotonicity in Lemma 7, where $W + W'$ is the pointwise summation of wallet states (e.g., if $W = \{\mathtt{A} \mapsto \{\mathtt{T} \mapsto 1\}\}$ and $W' = \{\mathtt{A} \mapsto \{\mathtt{T} \mapsto 2, \mathtt{T}' \mapsto 5\}\}$, then $W + W' = \{\mathtt{A} \mapsto \{\mathtt{T} \mapsto 3, \mathtt{T}' \mapsto 5\}\}$).

Lemma 7 (Wallet monotonicity). *Let X be valid in $W \mid \Gamma$. For all W'':*

$$W \mid \Gamma \xrightarrow{\mathsf{X}} W' \mid \Gamma' \quad \Longrightarrow \quad (W + W'') \mid \Gamma \xrightarrow{\mathsf{X}} (W' + W'') \mid \Gamma'$$

Proof. Recall that our contract model (Section 2) does not allow contracts to read users' wallets. Consequently, since X is valid in $W \mid \Gamma$, then X is also valid in $(W + W'') \mid \Gamma$ and it produces the same effect when executed in such state. □

We introduce in Definition 6 a relation $S \leq_\$ S'$ between blockchain states, which holds when S' can be obtained from S by enriching the users' wallets.

Definition 6. We write $S \leq_\$ S'$ when $S = W \mid \Gamma$ and $S' = (W + W') \mid \Gamma$, for some W, W' and Γ.

Lemma 8 states that the only user wallets that need to be taken into account to estimate the MEV are those of the adversary (Item 1). This follows because:

- $\mathcal{M}$ has no way to force other users to spend their tokens in the attack sequence, and

- contract functions cannot read users' wallets.

Furthermore, wealthier adversaries may potentially inflict more loss (Item 2).

Lemma 8 (MEV and adversaries' wallets). *We have that:*

1. *if* $\mathcal{M} \cap \operatorname{dom} W = \emptyset$, *then* $\mathrm{MEV}_D(S \mid W, C) = \mathrm{MEV}_D(S, C)$
2. *if* $S \preceq_\$ S'$, *then* $\mathrm{MEV}_D(S, C) \leq \mathrm{MEV}_D(S', C)$

Lemma 9 states that, for any contract state Γ, there exists a wallet state W yielding the same MEV as any of its enrichments $W' \geq_\$ W$. Together with Lemma 8, this ensures that an adversary $\mathcal{M}$ with a sufficiently rich wallet W can always inflict the maximum loss to the contracts in Γ.

Lemma 9 (MEV stability). *For all* C, D, *and* Γ:

$$\exists W.\ \forall W' \geq_\$ W.\ \ \mathrm{MEV}_D(W \mid \Gamma, C) = \mathrm{MEV}_D(W' \mid \Gamma, C)$$

Note that MEV stability implies the existence of the maximum MEV over all possible wallet states, i.e. $\max_{W''} \mathrm{MEV}_D(W'' \mid \Gamma, C)$. Such maximum is obtained taking $W'' = W$ since for any other W'' we have:

$$\begin{aligned} \mathrm{MEV}_D(W'' \mid \Gamma, C) &\leq \mathrm{MEV}_D((W + W'') \mid \Gamma, C) && \text{by Lemma 8(2)} \\ &= \mathrm{MEV}_D(W \mid \Gamma, C) && \text{by Lemma 9} \end{aligned}$$

By taking such maximum MEV we can then define a notion of MEV that does *not* depend on users' wallets (including $\mathcal{M}$'s). The new notion captures, for example, the attacks where the adversary can obtain the needed tokens through a *flash loan*, i.e. a loan of an arbitrary amount of tokens that must be entirely repaid within the same transaction [8,15].

Definition 7 (MEV of wealthy adversaries). For all C, D, Γ, we define:

$$\mathrm{MEV}^\infty_D(\Gamma, C) = \max_W \ \mathrm{MEV}_D(W \mid \Gamma, C)$$

The following lemma is the wealthy-adversaries counterpart of Lemma 2.

Lemma 10 (Basic properties of MEV^∞). *For all* Γ, Δ, *and* $C, D \subseteq \mathbb{A}_c$:

1. $\mathrm{MEV}^\infty_D(\Gamma, \emptyset) = \mathrm{MEV}^\infty_\emptyset(\Gamma, C) = 0$
2. $0 \leq \mathrm{MEV}^\infty_D(\Gamma, C) \leq \$_C(\Gamma)$
3. $\mathrm{MEV}^\infty_D(\Gamma, C) = \mathrm{MEV}^\infty_D(\Gamma, C \cap \dagger\Gamma) = \mathrm{MEV}^\infty_{D \cap \dagger\Gamma}(\Gamma, C)$
4. *if* $D \subseteq D'$, *then* $\mathrm{MEV}^\infty_D(\Gamma, C) \leq \mathrm{MEV}^\infty_{D'}(\Gamma, C)$
5. $\mathrm{MEV}^\infty_D(\Gamma, C) \leq \mathrm{MEV}^\infty_D(\Gamma \mid \Delta, C)$

Additionally, items 1 to 4 also hold for non well-formed states Γ.

The following Lemma is the wealthy-adversaries counterpart of Lemma 3: extending the state with new contracts does not affect the MEV extractable from the old contracts.

Lemma 11 (State narrowing). $\mathrm{MEV}^\infty_D(\Sigma \mid \Delta, C) = \mathrm{MEV}^\infty_D(\Sigma, C)$ *if* $D \subseteq \dagger\Sigma$.

Proof. By Definition 7 and Lemma 9 (twice), there exist W_0 and W_1 such that:

$$\forall W' \geq_\$ W_0.\ \mathrm{MEV}_D(W_0 \mid \Sigma \mid \Delta, C) = \mathrm{MEV}_D(W' \mid \Sigma \mid \Delta, C) = \mathrm{MEV}^\infty_D(\Sigma \mid \Delta, C)$$

$$\forall W' \geq_\$ W_1.\ \mathrm{MEV}_D(W_1 \mid \Sigma, C) = \mathrm{MEV}_D(W' \mid \Sigma, C) = \mathrm{MEV}^\infty_D(\Sigma, C)$$

Let $W = \max\{W_0, W_1\}$. By Lemma 3, since $D \subseteq \dagger\Sigma$:

$$\mathrm{MEV}_D(W \mid \Sigma \mid \Delta, C) \ = \ \mathrm{MEV}_D(W \mid \Sigma, C)$$

Since $W \geq_\$ W_0$ and $W \geq_\$ W_1$, by the above equations we have the thesis. □

Lemma 12 is the wealthy-adversaries counterpart of Lemma 4, as it gives sufficient conditions under which we can remove from the contract state Σ all the non-dependencies of C, preserving $\mathrm{MEV}^\infty_D(\Sigma, C)$. Compared to Lemma 4, here the conditions do not include the absence of token flows, since wealthy adversaries always have enough tokens to perform the optimal attack.

Lemma 12 (Callable contracts narrowing). *The equality:*

$$\mathrm{MEV}^\infty_D(\Sigma, C) \ = \ \mathrm{MEV}^\infty_{D \cap deps(C)}(\Sigma, C)$$

holds if the contracts in $\partial(C, D)$ *are sender-agnostic and* $\partial(C, D) \subseteq D$.

```
contract C0 {
  bool b;
  constructor(bool b_) { b=b_; }
  function f() { require (b||msg.sender==C1); T.send(msg.sender,5);}
}
contract C1 {
  function g() { C0.f(); T.send(msg.sender,5); }
}
```

Listing 5. Contracts for Examples 11.

Proof. By Lemma 9, it suffices to prove that, for a rich enough W:

$$\mathrm{MEV}_D(W \mid \Sigma, C) \;=\; \mathrm{MEV}_{D \cap deps(C)}(W \mid \Sigma, C)$$

This equation has the same form as the thesis of Lemma 4, whose proof we can replay almost verbatim. The only points of such proof which are changed are those where we exploit its "no tokens flows" assumptions to justify that the adversaries have enough tokens to perform their attack. Here, we have no similar assumption to exploit, but on the other hand we only have to prove the statement for *rich enough* W, so we can assume that W has enough tokens for the adversaries. □

The following examples shows that if the assumptions of Lemma 12 do not hold, then MEV preservation is not guaranteed.

Example 11. Consider the contracts in Listing 5. Let $\mathcal{M} = \{\mathtt{M}\}$, let $C = \{\mathtt{C0}\}$, $D = \{\mathtt{C1}, \mathtt{C0}\}$, and let $\Sigma = \mathtt{C0}[5 : \mathtt{T}, \mathtt{b} = \mathtt{false}] \mid \mathtt{C1}[0 : \mathtt{T}]$. We have that $\partial(C, D) = \{\mathtt{C0}\} \subseteq D$, but C0 violates sender-agnosticism since it enables token transfers only to C1. Therefore, Lemma 12 does not apply, and so MEV preservation is not guaranteed. Indeed, we have that:

$$\mathrm{MEV}_D(\Sigma, C) = 5 \cdot \$\mathbf{1}_{\mathrm{T}}$$

$$\mathrm{MEV}_{D \cap deps(C)}(\Sigma, C) = \mathrm{MEV}_{\{\mathtt{C0}\}}(\Sigma, C) = 0$$

The first equation is witnessed by the transaction $\mathtt{M} : \mathtt{C1.g}() \in \kappa_D(\mathcal{M})$, which extracts $5 : \mathtt{T}$ from C0. For the second equation, the MEV is zero since the only possible transaction $\mathtt{M} : \mathtt{C0.f}()$ reverts because of the require condition.

We now show that if the assumption $\partial(C, D) \subseteq D$ does not hold, then MEV preservation is not guaranteed either. Consider again the contracts in Listing 5, and let $\mathcal{M} = \{\mathtt{M}\}$, $C = \{\mathtt{C0}\}$, $D = \{\mathtt{C1}\}$, and $\Sigma = \mathtt{C0}[5 : \mathtt{T}, \mathtt{b} = \mathtt{true}] \mid \mathtt{C1}[0 : \mathtt{T}]$. This instance of C0 is sender-agnostic, but $C \not\subseteq D$. As before, we have that $\mathrm{MEV}_D(\Sigma, C) = 5 \cdot \$\mathbf{1}_{\mathrm{T}}$, but $\mathrm{MEV}_{D \cap deps(C)}(\Sigma, C) = \mathrm{MEV}_{\emptyset}(\Sigma, C) = 0$. ⋄

Theorem 13 is the wealthy-adversaries counterpart of Theorem 6. Compared to that, the assumption about the absence of indirect token flows is not required.

Theorem 13 (MEV^∞ and observational invariance)**.** *For any state* $\Sigma \mid \Delta$ *(not necessarily well-formed),*

$$\mathrm{MEV}^\infty_D(\Sigma \mid \Delta, \dagger\Delta) \;=\; \mathrm{MEV}^\infty_{D \cap \dagger\Delta}(\Sigma \mid \Delta, \dagger\Delta)$$

holds iff Δ *is observationally invariant w.r.t. adversarial moves in* $W \mid \Sigma$*, for all sufficiently large* W*.*

Proof. Assume that observational invariance holds for all $W \geq_\$ W_{oi}$. We show:

$$\mathrm{MEV}^\infty_{D \cap \dagger\Delta}(\Sigma \mid \Delta, \dagger\Delta) \geq \mathrm{MEV}^\infty_D(\Sigma \mid \Delta, \dagger\Delta) \tag{4}$$

since the other inequality is guaranteed by Lemma 10(4). To do so, we consider a W_{max} that maximizes $\mathrm{MEV}_D(W_{max} \mid \Sigma \mid \Delta, \dagger\Delta)$. W.l.o.g., by MEV stability (Lemma 9) we can assume $W_{max} \geq W_{oi}$.

Then, to prove (4) it suffices to find a wallet W' satisfying the following:

$$\mathrm{MEV}^\infty_{D \cap \dagger\Delta}(\Sigma \mid \Delta, \dagger\Delta) \geq \mathrm{MEV}_{D \cap \dagger\Delta}(W' \mid \Sigma \mid \Delta, \dagger\Delta) \tag{5}$$

$$\geq \mathrm{MEV}_D(W_{max} \mid \Sigma \mid \Delta, \dagger\Delta) \tag{6}$$

$$= \mathrm{MEV}^\infty_D(\Sigma \mid \Delta, \dagger\Delta) \tag{7}$$

where inequality (5) follows from the definition of MEV^∞, which takes the maximum over all possible wallets W', and equality (7) follows from our choice of W_{max} as the wallet state that gives MEV^∞.

For inequality (6), we proceed similarly to the proof of Theorem 6, where we show that for any MEV-extracting sequence of transactions in $W_{max} \mid \Sigma \mid \Delta$ there exists another sequence of transactions that extracts the same value but targeting only contracts in Δ. The same steps of our previous proof, with one exception, can be replayed since observational invariance holds by assumption and $W_{max} \geq W_{oi}$. The exception is due to those steps that exploited the absence of token flows (an assumption of Theorem 6) to justify that the new sequence has enough funds. Here, we do not have that hypothesis. Nonetheless, we can still claim that the new sequence is well funded since we can simply take W' to include all the tokens we need for that purpose. Indeed, wealthy adversaries can always provide funds for their attacks. □

5. MEV non-interference

We formalise secure contract composability as a relation between blockchain states S and contract states Δ: intuitively, $S \not\rightsquigarrow \Delta$ means that the adversary cannot leverage S to inflict more MEV to Δ than it would be possible by interacting with Δ alone. We will then say that S is *MEV non-interfering* with Δ.

5.1. Definition

Formally, MEV non-interference $S \not\rightsquigarrow \Delta$ 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. We write $S \rightsquigarrow \Delta$ when $S \not\rightsquigarrow \Delta$ does not hold.

Definition 8 (MEV non-interference)**.** A state S is MEV non-interfering with Δ, in symbols $S \not\rightsquigarrow \Delta$, when $\mathrm{MEV}(S \mid \Delta, \dagger\Delta) = \mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta)$.

Note that the inequality $\geq$ in Definition 8 always holds by Lemma 2(4), so checking MEV non-interference only requires to check $\leq$.

The following example discriminates MEV non-interference from Babel *et al.*' ε-composability, showing that a contract with intended MEV but no interactions with the context enjoys MEV non-interference, but it is not ε-composable.

Example 12. Consider the airdrop contract in Listing 3, let S be any blockchain state, and let $\Delta = \mathtt{Airdrop}[n : \mathtt{T}, \mathtt{t} = \mathtt{T}]$. We have that $S \not\rightsquigarrow \Delta$, since:

$$\mathrm{MEV}(S \mid \Delta, \{\mathtt{Airdrop}\}) \;=\; n \cdot \$\mathbf{1}_{\mathtt{T}} \;=\; \mathrm{MEV}_{\{\mathtt{Airdrop}\}}(S \mid \Delta, \{\mathtt{Airdrop}\})$$

Here, MEV non-interference correctly reflects the fact that $\mathcal{M}$ cannot leverage the contracts in S to damage the airdrop. ◇

```
contract D {
  function f() { require (C.get()); T.send(msg.sender,1); }
}
contract C {
  bool x; uint fee;
  constructor(uint fee) { x=false; fee=fee_; }
  function get() { return x; }
  function set() { T.receive(fee); x=true; }
}
```

Listing 6. Contracts for Examples 13 and 17.

We show some compositions that do not enjoy MEV non-interference, for different causes. In Example 13, the interference is caused by contract dependencies, while in Example 14 it is caused by token flows.

Example 13. Consider the contracts in Listing 6, and let:

$$S = \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{C}[0 : \mathtt{T}, \mathtt{x} = \mathtt{false}, \mathtt{fee} = 0] \qquad \Delta = \mathtt{D}[100 : \mathtt{ETH}]$$

We have that $\mathrm{MEV}(S \mid \Delta, \{\mathtt{D}\}) = 100 \cdot \$\mathbf{1}_{\mathtt{ETH}}$, while $\mathrm{MEV}_{\{\mathtt{D}\}}(S \mid \Delta, \{\mathtt{D}\}) = 0$, since $\mathtt{M}$ cannot call $\mathtt{C.set()}$. Therefore, $S \rightsquigarrow \Delta$. ◇

Example 14. Consider the contracts in Listings 2 and 3, and let:

$$\begin{aligned} S &= \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{PriceOracle}[\mathtt{fee} = 0, \mathtt{prices} = \{\mathtt{T}_1 = 10, \mathtt{T}_2 = 1\}, \ldots] \\ \Gamma &= \mathtt{Airdrop}[1 : \mathtt{T}_1, \mathtt{t} = \mathtt{T}_1] \\ \Delta &= \mathtt{Exchange}[0 : \mathtt{T}_1, 10 : \mathtt{T}_2, \mathtt{t0} = \mathtt{T}_1, \mathtt{t1} = \mathtt{T}_2] \end{aligned}$$

Assume that $\$\mathbf{1}_{\mathtt{T}_1} = \$\mathbf{1}_{\mathtt{T}_2}$, and let $\mathcal{M} = \{\mathtt{M}\}$. We have that:

$$\mathrm{MEV}(S \mid \Gamma \mid \Delta, \{\mathtt{Exchange}\}) = 10 \cdot \$\mathbf{1}_{\mathtt{T}_2} - 1 \cdot \$\mathbf{1}_{\mathtt{T}_1} = 9 \cdot \$\mathbf{1}_{\mathtt{T}_2}$$

since $\mathtt{M}$ can first extract $1 : \mathtt{T}_1$ from the airdrop, and then swap it for $10 : \mathtt{T}_2$ through the $\mathtt{Exchange}$. Instead, $\mathrm{MEV}_{\{\mathtt{Exchange}\}}(S \mid \Gamma \mid \Delta, \{\mathtt{Exchange}\}) = 0$, since $\mathtt{M}$ cannot obtain the tokens $\mathtt{T}_1$ needed to use the $\mathtt{Exchange}$. Then, $S \mid \Gamma \rightsquigarrow \Delta$. ◇

The term *MEV non-interference* is inspired by classical non-interference in information-flow security [9]. There, one typically partitions inputs into public and private components and requires that changes to private inputs do not affect publicly observable outputs. In our setting, the surrounding blockchain state (and the adversary's ability to interact with it) plays the role of the external input, while the economic loss suffered by a designated set of victims (the newly deployed contracts) is the observable outcome that we seek to protect. Similarly, MEV non-interference requires that giving the adversary access to the pre-existing blockchain state does not increase the value that can be extracted from the victim contracts. Thus, although the surrounding state may change the adversary's available strategies, it must not worsen the economic outcome of the newly deployed contracts.

Like classic non-interference, also our notion of MEV non-interference is qualitative. Approximate or quantitative variants could also in principle be defined, e.g., by considering the difference between the MEVs in Definition 8. While quantitative variants could also be useful in practice, our qualitative notion appears to be simpler, so we start our investigation from that, leaving the other notions for future work (see Section 8).

5.2. Sufficient conditions for MEV non-interference

Theorem 14 gives sufficient conditions for MEV non-interference. Condition 1 states that $S \not\rightsquigarrow \Delta$ holds whenever the new contracts Δ are MEV-free: a special case is when Δ has no tokens, by Lemma 2. Condition 2 allows to establish $S \not\rightsquigarrow \Delta$ even when Δ has MEV, provided that the token flows from S to Δ cannot be exploited by adversaries, and that Δ is observationally invariant w.r.t. adversarial moves in S. Recall that this property (see Definition 5) ensures that any transaction sent by adversary to the contracts in S is not observable by the contracts in Δ, and therefore the ability to call the contracts in S does not give the adversary any advantage compared to calling only Δ. For instance, condition 2 is applicable to establish MEV non-interference in Example 12, since the `Airdrop` contract is (trivially) observationally invariant w.r.t. any S.

Theorem 14 (Sufficient conditions for $\not\rightsquigarrow$)**.** *Let $S \mid \Delta$ be a blockchain state, not necessarily well-formed. MEV non-interference $S \not\rightsquigarrow \Delta$ holds if any of the following conditions is satisfied:*

1. $\mathrm{MEV}(S \mid \Delta, \dagger\Delta) = 0$
2. *There is no token flow from S to Δ in $S \mid \Delta$, and Δ is observationally invariant in S.*

Proof. For condition 1 we have:

$$\begin{aligned} 0 &\le \mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) && \text{by Lemma 2(2)} \\ &\le \mathrm{MEV}(S \mid \Delta, \dagger\Delta) && \text{by Lemma 2(4)} \\ &= 0 && \text{by hypothesis} \end{aligned}$$

which implies $\mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) = 0$, and therefore $S \not\rightsquigarrow \Delta$.

To prove that $S \not\rightsquigarrow \Delta$ under condition 2, observe that the equality:

$$\mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) = \mathrm{MEV}(S \mid \Delta, \dagger\Delta)$$

follows directly from Theorem 6. □

We have already shown in Example 12 a case where MEV non-interference does not imply condition 1 of Theorem 14. It is easy to show a case where MEV non-interference holds but condition 2 does not hold. For example, consider a contract `D` with a state variable that can be set to the value of some variable in a contract in S, but does not otherwise affect the MEV extractable from `D`. Then, observational invariance does not hold, but MEV non-interference holds.

We will see several applications of Theorem 14 laster in Section 7.

5.3. Resistance to front-running attacks

In general, $S \not\rightsquigarrow \Delta$ does not imply $S \mid \Gamma \not\rightsquigarrow \Delta$. For instance, in Example 14 we showed that $S \mid \Gamma \rightsquigarrow \Delta$, but if we remove the `Airdrop` contract then `M` is no longer able to extract MEV from the `Exchange`, since `M`'s wallet is empty: hence, $S \not\rightsquigarrow \Delta$. This means that, in general, MEV non-interference is not robust against attacks where adversaries manage to deploy contracts Γ *before* Δ. Namely, assume that a user detects that $S \not\rightsquigarrow \Delta$, and then sends a transaction to deploy the contracts Δ. If this transaction is front-run with another transaction that deploys Γ, then the new state $S \mid \Gamma$ could MEV-interfere with Δ.

Theorem 15 establishes sufficient conditions under which MEV non-interference $S \not\rightsquigarrow \Delta$ is preserved when the deployment of Δ is front-run with the deployment of Γ. Intuitively, the presence of Γ should be "unobservable" from Δ: roughly, this is guaranteed by requiring that Δ or its dependencies cannot perform actions depending on the presence of Γ, and Γ cannot transfer tokens to the dependencies of Δ. More precisely, condition 1 requires the contracts in Δ to be *sender-agnostic*, i.e., their behaviour must not depend on the identity of the caller. Condition 2 requires that the deployment of Γ does not enable new token flows into Δ. If either condition is violated, contracts in Γ may enable forms of token extraction that were impossible when Δ interacted only with S. For instance, if a contract in Δ is not sender-agnostic, an access control check can succeed when the caller is a contract in Γ (see Example 15). Similarly, token transfers from Γ to Δ may unlock MEV-extraction opportunities from Δ (see Example 16).

Theorem 15 (Front-running resistance)**.** $S \not\rightsquigarrow \Delta \iff S \mid \Gamma \not\rightsquigarrow \Delta$ *holds if*

1. *the contracts in $deps(\Delta)$ are sender-agnostic, and*
2. *there are no token flows from $\dagger(S \mid \Gamma) \setminus deps(\Delta)$ to $deps(\Delta)$ in $S \mid \Gamma \mid \Delta$.*

Proof. The thesis can be written as:

$$\begin{array}{ccc} \mathrm{MEV}(S \mid \Delta, \dagger\Delta) & = & \mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) \\ & \iff & \\ \mathrm{MEV}(S \mid \Gamma \mid \Delta, \dagger\Delta) & = & \mathrm{MEV}_{\dagger\Delta}(S \mid \Gamma \mid \Delta, \dagger\Delta) \end{array}$$

We prove the thesis by establishing the following two equalities:

$$\mathrm{MEV}(S \mid \Delta, \dagger\Delta) = \mathrm{MEV}(S \mid \Gamma \mid \Delta, \dagger\Delta) \tag{8}$$

$$\mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) = \mathrm{MEV}_{\dagger\Delta}(S \mid \Gamma \mid \Delta, \dagger\Delta) \tag{9}$$

We start by observing that, since $S \mid \Gamma$ and $S \mid \Delta$ are well-formed, then Γ has no dependencies in Δ and Δ has no dependencies in Γ. Then, the states $S \mid \Gamma \mid \Delta$ and $S \mid \Delta \mid \Gamma$ are equivalent:

$$S \mid \Gamma \mid \Delta \;=\; S \mid \Delta \mid \Gamma \tag{10}$$

For (8), note that by Lemma 2(3), if we define $D = \dagger(S \mid \Gamma \mid \Delta)$ we have:

$$\mathrm{MEV}(S \mid \Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}_D(S \mid \Gamma \mid \Delta, \dagger\Delta) \tag{11}$$

Using such D, the conditions of Theorem 5 are satisfied:

1. the contracts in $deps(\Delta)$ are sender-agnostic by hypothesis 1;
2. $deps(\Delta) \subseteq D$ holds trivially;
3. there are no token flows from $deps(D) \setminus deps(\Delta)$ to $deps(\Delta)$ in $S \mid \Gamma \mid \Delta$, by hypothesis 2.

Therefore, we obtain (8) from the chain of (in)equalities:

$$\begin{array}{lll} \mathrm{MEV}(S \mid \Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}_D(S \mid \Gamma \mid \Delta, \dagger\Delta) & & \text{by (11)} \\ = \mathrm{MEV}_{D \cap deps(\Delta)}(S \mid \Delta, \dagger\Delta) & & \text{by Theorem 5} \\ = \mathrm{MEV}_{deps(\Delta)}(S \mid \Delta, \dagger\Delta) & & \text{since } deps(\Delta) \subseteq D \\ \leq \mathrm{MEV}(S \mid \Delta, \dagger\Delta) & & \text{by Lemma 2(4)} \\ \leq \mathrm{MEV}(S \mid \Delta \mid \Gamma, \dagger\Delta) & & \text{by Lemma 2(5)} \\ = \mathrm{MEV}(S \mid \Gamma \mid \Delta, \dagger\Delta) & & \text{by (10)} \end{array}$$

For (9), we have that:

$$\begin{array}{lll} \mathrm{MEV}_{\dagger\Delta}(S \mid \Delta, \dagger\Delta) = \mathrm{MEV}_{\dagger\Delta}(S \mid \Delta \mid \Gamma, \dagger\Delta) & & \text{by Lemma 3, as } \dagger\Delta \subseteq \dagger(S \mid \Delta) \\ = \mathrm{MEV}_{\dagger\Delta}(S \mid \Gamma \mid \Delta, \dagger\Delta) & & \text{by (10)} \end{array}$$

□

```
contract D {
  function f() { require (Lock.b()); T.send(msg.sender,1); }
}
contract C {
  function g() { Lock.unlock(); }
}
contract Lock { // not sender-agnostic
  bool public b=false;
  function unlock() { require (msg.sender==C); b=true; }
}
```

Listing 7. Contracts for Example 15.

Example 15. Consider the contracts in Listing 7, let $\mathcal{M} = \{\mathtt{M}\}$, and let:

$$S = \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{Lock}[\mathtt{b} = \mathtt{false}] \qquad \Gamma = \mathtt{C}[0 : \mathtt{T}] \qquad \Delta = \mathtt{D}[1 : \mathtt{T}]$$

In $S \mid \Delta$, it is not possible to extract the $1 : \mathtt{T}$ from $\mathtt{D}$, since $\mathtt{M}$ cannot set `Lock.b`. Therefore, $\mathrm{MEV}(S \mid \Delta, \{\mathtt{D}\}) = 0$, which implies $S \not\rightsquigarrow \Delta$. Instead, in $S \mid \Gamma \mid \Delta$, if $\mathtt{M}$ has unrestricted access to $\mathtt{C}$, then she can set `Lock.b` and then extract $1 : \mathtt{T}$ from $\mathtt{D}$. When the set of callable contracts is restricted to $\{\mathtt{D}\}$, instead, the MEV extractable from $\mathtt{D}$ is zero. Therefore, $S \mid \Gamma \not\rightsquigarrow \Delta$. Note that the contract $\mathtt{Lock} \in \mathit{deps}(\Delta)$ is *not* sender-agnostic, hence Theorem 15 does not apply. ◇

```
contract D {
  function f() { require (PaidLock.b()); T.send(msg.sender,1); }
}
contract C {
  function g() { T.approve(PaidLock,1); PaidLock.unlock(); }
}
contract PaidLock { // token flow from C
  bool public b=false;
  function unlock() { T.receive(1); b=true; }
}
```

Listing 8. Contracts for Example 16.

Example 16. Consider the contracts in Listing 8, let $\mathcal{M} = \{\mathtt{M}\}$, and let:

$$S = \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{PaidLock}[\mathtt{b} = \mathtt{false}] \qquad \Gamma = \mathtt{C}[1 : \mathtt{T}] \qquad \Delta = \mathtt{D}[1 : \mathtt{T}]$$

The reasoning is similar to the in Example 15. We have $S \not\rightsquigarrow \Delta$ because, without the token in $\mathtt{C}$, it is not possible to unlock the extraction of $1 : \mathtt{T}$ from $\mathtt{D}$. Instead, $S \mid \Gamma \not\rightsquigarrow \Delta$, since $\mathtt{M}$ can call $\mathtt{C}$ to set `PaidLock.b` and then extract $1 : \mathtt{T}$ from $\mathtt{D}$. Unlike Example 15, all contracts are sender-agnostic, but there is a token flow from $\mathtt{C}$ to $\mathtt{PaidLock} \in \mathit{deps}(\Delta)$, so Theorem 15 does not apply. ◇

6. MEV non-interference for wealthy adversaries

MEV non-interference $S \not\rightsquigarrow \Delta$ may or may not hold depending on the wealth of the adversary in S, as we have already observed in Lemma 8 that MEV depends on the adversary's wallets. As noted before, assuming bounds on the capital available to the adversary may be unsafe, and consequently we introduced in Definition 7 a notion of MEV that assumes an unbounded adversary's wealth. We now introduce a wealthy-adversaries counterpart to MEV non-interference: $\Gamma \not\rightsquigarrow^{\infty} \Delta$ holds when contracts Γ do not interfere with the MEV extractable from Δ *regardless* of the adversary's wealth.

Definition 9 (MEV non-interference against wealthy adversaries)**.** A contract state Γ is MEV^{∞} *non-interfering* with Δ, in symbols $\Gamma \not\rightsquigarrow^{\infty} \Delta$, when

$$\mathrm{MEV}^{\infty}(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}^{\infty}_{\dagger\Delta}(\Gamma \mid \Delta, \dagger\Delta)$$

Lemma 16 characterizes $\not\rightsquigarrow^{\infty}$ in terms of $\not\rightsquigarrow$: intuitively, $\Gamma \not\rightsquigarrow^{\infty} \Delta$ holds whenever $W \mid \Gamma \not\rightsquigarrow \Delta$ holds for rich enough adversaries' wallets.

Proposition 16. *$\Gamma \not\rightsquigarrow^{\infty} \Delta$ if and only if $\exists W_0.\ \forall W \geq_{\$} W_0.\ W \mid \Gamma \not\rightsquigarrow \Delta$.*

Of course, we cannot deduce $\Gamma \not\rightsquigarrow^{\infty} \Delta$ when $W \mid \Gamma \not\rightsquigarrow \Delta$ holds for *some* W: indeed, a "poor" adversary could not be able to call some MEV-triggering method in Γ, while a wealthy one could: then, we could observe a discrepancy between the restricted and unrestricted MEV^{∞} which was not visible for MEV. The following example elaborates on this informal argument.

Example 17. The implication "if $W \mid \Gamma \not\rightsquigarrow \Delta$ then $\Gamma \not\rightsquigarrow^{\infty} \Delta$" does not hold. To show that, consider again the contracts in Listing 6, and let:

$$W = \mathtt{M}[0 : \mathtt{T}] \qquad \Gamma = \mathtt{C}[0 : \mathtt{T}, \mathtt{x} = \mathtt{false}, \mathtt{fee} = 1] \qquad \Delta = \mathtt{D}[100 : \mathtt{ETH}]$$

For $\not\rightsquigarrow$, we have that $\mathrm{MEV}(W \mid \Gamma \mid \Delta, \{\mathtt{D}\}) = 0$, since $\mathtt{M}$ does not have the $1 : \mathtt{T}$ needed to extract the $\mathtt{ETH}$ from Δ. Therefore, $W \mid \Gamma \not\rightsquigarrow \Delta$. For $\not\rightsquigarrow^{\infty}$, we have that $\mathrm{MEV}^{\infty}(\Gamma \mid \Delta, \{\mathtt{D}\}) = 100 \cdot \$\mathbf{1}_{\mathtt{ETH}}$, while $\mathrm{MEV}^{\infty}_{\{\mathtt{D}\}}(\Gamma \mid \Delta, \{\mathtt{D}\}) = 0$, since $\mathtt{M}$ cannot call the `set` method of $\mathtt{C}$. Then, $\Gamma \not\rightsquigarrow^{\infty} \Delta$. ◇

The following theorem refines Theorem 14, giving sufficient conditions for $\not\rightsquigarrow^{\infty}$. Note that the absence of token flows is no longer required.

Theorem 17 (Sufficient conditions for $\not\rightsquigarrow^{\infty}$)**.** *For any state $\Gamma \mid \Delta$ (not necessarily well-formed), each of the following conditions implies $\Gamma \not\rightsquigarrow^{\infty} \Delta$:*

1. $\mathrm{MEV}^{\infty}(\Gamma \mid \Delta, \dagger\Delta) = 0$*;*
2. *$\mathit{deps}(\Delta)$ and $\dagger\Gamma$ are disjoint;*

Table 2
MEV non-interference of DeFi compositions.

Dependencies Γ	New contracts Δ	$W \mid \Gamma \not\rightsquigarrow \Delta$	$\Gamma \not\rightsquigarrow^{\infty} \Delta$
`IPriceOracle` | ⋯	`Exchange`	✗	✗
`PriceOracle` | ⋯	`Exchange`	✗	✓[3]
⋯	`AMM`	✗	✓[2]
`AMM`	`Option`	✗	✗
`PriceOracle`	`Option`	✓[2]	✓[3]
`IExchange` | `IExchange`	`SwapProxy`	✓[1]	✓[1]
`IExchange` | `IExchange`	`SwapRouter`	✓[1]	✓[1]
`PriceOracle` | `LinearInterestRate`	`LP`	✓[2]	✓[3]
`AMM` | `IInterestRate`	`LP`	✗	✗
`IExchange` | `IExchange` | `ILoaner`	`LPArbitrage`	✓[1]	✓[1]
`IExchange` | `IExchange` | `IFlashLoaner`	`FlashLoanArbitrage`	✓[1]	✓[1]

3. Δ *is observationally invariant in* $W \mid \Gamma$*, for all sufficiently large* W*.*

Proof. For Theorem 1, $\Gamma \not\rightsquigarrow^{\infty} \Delta$ is implied by:

$$\begin{aligned} 0 &\le \mathrm{MEV}^{\infty}_{\dagger\Delta}(\Gamma \mid \Delta, \dagger\Delta) && \text{by Lemma 10(2)} \\ &\le \mathrm{MEV}^{\infty}(\Gamma \mid \Delta, \dagger\Delta) && \text{by Lemma 10(4)} \\ &= 0 && \text{by hypothesis} \end{aligned}$$

For Theorem 2, assume that $deps(\Delta)$ and $\dagger\Gamma$ are disjoint. We prove that Δ is observationally invariant in $W \mid \Gamma$, for all W. Definition 5 requires checking the effects of all transactions $\mathtt{Y}$ fired in $W \mid \Gamma \mid \Delta$ which trigger a call $\mathtt{D} : \mathtt{C.f(args)}$ with $\mathtt{D} \in \dagger\Delta$ and $\mathtt{C} \in \dagger(W \mid \Gamma) = \dagger\Gamma$. This implies that $\mathtt{C} \in deps(\Delta) \cap \dagger\Gamma$, contradicting the hypothesis in Theorem 2.

Since the assumption of Theorem 2 implies that of Theorem 3, we only need to prove the theorem in the latter case. Under this assumption, the thesis follows directly from Theorem 13. □

Theorem 18 is the wealthy-adversaries counterpart of Theorem 15, establishing that MEV non-interference is robust against front-running the deployment of Δ with adversarial contracts Γ whenever the dependencies of Δ are sender-agnostic. Intuitively, this holds because any MEV extraction exploiting Γ can also be performed by directly calling the dependencies of Δ, since they are not influenced by the caller's identity.

Theorem 18 (Front-running resistance). $\Sigma \not\rightsquigarrow^{\infty} \Delta \iff \Sigma \mid \Gamma \not\rightsquigarrow^{\infty} \Delta$ *holds if the contracts in* $deps(\Delta)$ *are sender-agnostic.*

Proof. We replay the proof of Theorem 15, replacing $\not\rightsquigarrow$ with its wealthy-adversary counterpart $\not\rightsquigarrow^{\infty}$, and exploiting the wealthy-adversary counterparts of the used lemmas about MEV. More in detail, each use of Lemma 4 within the proof of Theorem 15 is replaced here with Lemma 12, which only requires the sender-agnostic assumption on $deps(\Delta)$. □

7. Application to DeFi compositions

We now analyse MEV non-interference of DeFi compositions, considering as basic building blocks typical DeFi protocols. We summarize the results of the analysis in Table 2, where each line shows a compound contract Δ, the context Γ, and whether $\not\rightsquigarrow$ and $\not\rightsquigarrow^{\infty}$ hold (✓) or not (✗). In the table we omit the actual wallet and contract states, with the following interpretation:

- ✓ in the $W \mid \Gamma \not\rightsquigarrow \Delta$ column means that non-interference holds for all W, and for all Γ and Δ of the given form; ✗ means that $\not\rightsquigarrow$ fails to hold for some W, and for some Γ and Δ of the given form;
- ✓ in the $\Gamma \not\rightsquigarrow^{\infty} \Delta$ column means that non-interference holds for all Γ and Δ of the given form; ✗ means that $\not\rightsquigarrow^{\infty}$ fails to hold for some Γ and Δ of the given form.

Entries marked ✓ follow by Theorems 14 and 17, and indicate which case of the respective theorem applies. Note that these theorems do not require the well-formedness assumption, i.e. they also apply to systems with circular dependencies. This is the case, e.g., of the `FlashLoanArbitrage`, which uses a flash loan mechanism to obtain the liquidity needed to perform an arbitrage between two exchanges. Instead, all the other compositions in Table 2 are well-formed. For entries marked ✗ we provide a counterexample in the respective section. Note that the ✓ and ✗ results may still hold when Γ is extended with further contracts, whenever the conditions of Theorem 15 and 18 are satisfied.

7.1. Exchange

The `Exchange` contract in Listing 2 does not enjoy neither MEV nor MEV$^{\infty}$ non-interferences w.r.t. a generic context. For example, MEV interference may occur when the context includes a contract from which the adversary can receive tokens needed to interact with the `Exchange`, e.g. an `Airdrop` or another `Exchange`. Instead, MEV$^{\infty}$ interference may occur when the `IPriceOracle` instance upon which the `Exchange` depends can be manipulated by the adversary. This happens e.g. when the price oracle is an `AMM`.

When, instead, the Exchange depends on the PriceOracle contract in Listing 2, it is observationally invariant w.r.t. adversarial moves in the context. Therefore, MEV$^\infty$ non-interference holds by Theorem 17(3). Note that MEV non-interference may still fail because of token flows to the adversary.

7.2. AMM

We specify in Listing 9 an Automated Market Maker (AMM) inspired by Uniswap v2 [18–20]. The AMM stores reserves of two token types $\mathtt{T}_0$ and $\mathtt{T}_1$; these types and their initial reserves are defined upon deployment. The method swap allows anyone to exchange units of $\mathtt{T}_0$ for units of $\mathtt{T}_1$ and vice-versa, provided that the product between their reserves remains constant. More specifically, a transaction $\mathtt{A} : \mathtt{swap}(\mathtt{T}_0, x, y_{\min})$ allows A to send $x : \mathtt{T}_0$ to the AMM, and receive at least $y_{\min} : \mathtt{T}_1$ in exchange. Symmetrically, $\mathtt{A} : \mathtt{swap}(\mathtt{T}_1, x, y_{\min})$ allows A to exchange $x : \mathtt{T}_1$ for at least $y_{\min} : \mathtt{T}_0$. The method getRate gives the exchange rate between the two token types, thereby making the AMM usable as a price oracle.

AMMs have a well-known source of MEV when the internal exchange rate given by getRate is not aligned with the external rate (i.e., the ratio of the token prices given by the price function $\$\mathbf{1}$). In this case, MEV can be extracted by firing a swap transaction that realigns the internal with the external rate [21].

```
contract AMM is IPriceOracle, IExchange {
  token immutable public t0; token immutable public t1;

  constructor(token t0_, uint x0, token t1_, uint x1) {
    t0=t0_; t1=t1_; require t0!=t1 && x0>0 && x1>0;
    t0.receive(x0); t1.receive(x1);
  }
  // exchange rate to sell ti and buy to (multiplied by 1e6)
  function getRate(token ti, token to) public returns(uint) {
    require (ti==t0 && to==t1) || (ti==t1 && to==t0);
    uint ri = ti.balanceOf(this);
    uint ro = to.balanceOf(this);
    return (1_000_000 * ro)/ri; // note ro,ri flipped wrt Exchange
  }
  // list of tokens swappable through the AMM
  function getTokens() public view returns (token[] memory) {
    token[] memory ts = new token[](2); ts[0]=t0; ts[1]=t1; return ts;
  }
  // swaps x:ti for at least ymin:to
  function swap(token ti, uint x, token to, uint ymin) public returns (uint) {
    require (ti==t0 && to==t1) || (ti==t1 && to==t0);
    ti.receive(x);
    uint y = (x * getRate(ti,to))/1_000_000;
    require (y>=ymin);
    to.send(msg.sender,y);
    return y;
  }
}
```

Listing 9. A constant-product AMM contract.

To study MEV non-interference, consider two AMM instances for the pairs $(\mathtt{T}_0, \mathtt{T}_1)$ and $(\mathtt{T}_1, \mathtt{T}_2)$. Let $\mathcal{M} = \{\mathtt{M}\}$, let $\$\mathbf{1}_{\mathtt{T}_0} = \$\mathbf{1}_{\mathtt{T}_1} = \$\mathbf{1}_{\mathtt{T}_2} = 1$, and let:

$$S = \mathtt{M}[3 : \mathtt{T}_0] \mid \mathtt{AMM1}[6 : \mathtt{T}_0, 6 : \mathtt{T}_1] \qquad \Delta = \mathtt{AMM2}[4 : \mathtt{T}_1, 9 : \mathtt{T}_2]$$

The internal exchange rate for selling $\mathtt{T}_1$ and buying $\mathtt{T}_2$ in AMM2 is $4/9$, while the external rate is 1: therefore, it is possible to extract MEV from AMM2. Since M does not have any units of $\mathtt{T}_1$ that are needed to swap through AMM2, extracting such MEV requires first obtaining the needed units through AMM1. The sequence of transactions that maximises the loss of AMM2 is the following:

$$S \xrightarrow{\mathtt{M}:\mathtt{AMM1.swap}(\mathtt{T}_0,3,0)} \mathtt{M}[2 : \mathtt{T}_1] \mid \mathtt{AMM1}[9 : \mathtt{T}_0, 4 : \mathtt{T}_1] \mid \mathtt{AMM2}[4 : \mathtt{T}_1, 9 : \mathtt{T}_2]$$

$$\xrightarrow{\mathtt{M}:\mathtt{AMM2.swap}(\mathtt{T}_1,2,0)} \mathtt{M}[3 : \mathtt{T}_2] \mid \mathtt{AMM1}[9 : \mathtt{T}_0, 4 : \mathtt{T}_1] \mid \mathtt{AMM2}[6 : \mathtt{T}_1, 6 : \mathtt{T}_2]$$

This causes AMM2 to receive $2 : \mathtt{T}_1$ and send $3 : \mathtt{T}_2$, for an overall loss of 1. Therefore, $\mathrm{MEV}(S, \{\mathtt{AMM2}\}) = 1$. Instead, when M is restricted to use $D = \{\mathtt{AMM2}\}$, it has no way to obtain the tokens $\mathtt{T}_1$ that are needed to extract value from AMM2: therefore, $\mathrm{MEV}_{\{\mathtt{AMM2}\}}(S, \{\mathtt{AMM2}\}) = 0$. Summing up, $S \nrightsquigarrow \Delta$. Similar attacks are possible whenever the context allows the adversary to obtain tokens needed to interact with the AMM in Δ.

For wealthy adversaries, instead, AMMs are always MEV non-interfering with any context. This holds because wealthy adversaries always have enough tokens to attack the new AMM in Δ, without the need of exploiting the context Γ. This follows from Theorem 17(2).

```
contract Option {
  address owner;   // option seller
  address buyer;   // option holder
  token t;         // underlying asset
  uint strike;     // strike price (multiplied by 1_000_000)
  uint expiry;     // expiration time
  uint fee;        // option price paid by buyer
  uint payout;     // fixed payoff if strike reached
  IPriceOracle public oracle;

  constructor(token t_,uint s,uint e,uint f,uint p,IPriceOracle o) {
    t=t_; strike=s; expiry=e; fee=f; payout=p; oracle=o; owner=msg.sender;
    ETH.receive(payout);
    require (expiry>block.number);
  }

  function buy() public {
    require (buyer==address(0)); // option has not been sold yet
    ETH.receive(fee);
    ETH.send(owner,fee);
    buyer=msg.sender;
  }

  function exercise() public {
    require (block.number<=expiry && buyer!=address(0) && msg.sender==buyer);
    uint price=oracle.getRate(t,ETH);
    require (price>=strike);
    ETH.send(buyer,payout);
  }

  function reclaim() public {
    require (block.number>expiry && msg.sender==owner);
    ETH.send(owner,payout);
  }
}
```

Listing 10. A binary option contract relying on an external price oracle.

7.3. Binary option

The `Option` contract in Listing 10 implements a minimal american binary call option settled via a price oracle. A seller deploys the contract by locking a payoff in `ETH`, and choosing a target token `T` and a strike price it has to reach in order to claim the option. A buyer may then acquire the option by paying a fixed fee. Before expiration, if the price of the token `T` is greater than or equal to the strike price, the buyer can redeem the payoff. Otherwise, the payoff is returned to the seller after expiration.

Consider an instance of `Option` using an `AMM` as price oracle, with payout of $10 : \mathtt{ETH}$, fee of $1 : \mathtt{ETH}$, owner `A`, target token `T`, and strike price $2 \cdot 10^6$:

$$S = \mathtt{M}[301 : \mathtt{ETH}] \mid \mathtt{AMM}[600 : \mathtt{ETH}, 600 : \mathtt{T}] \mid \mathtt{block.number} = n - k \mid \cdots$$

$$\Delta = \mathtt{Option}[10 : \mathtt{ETH}, \mathtt{owner} = \mathtt{A}, \mathtt{buyer} = 0, \mathtt{t} = \mathtt{T}, \mathtt{strike} = 2\mathrm{e}6, \mathtt{expiry} = n, \mathtt{fee} = 1, \ldots]$$

Note that the current exchange rate is $1 \cdot 10^6$, while exercising the option requires to make it exceed $2 \cdot 10^6$. Since the option has not expired ($\mathtt{block.num} = n - k < n$) and the adversary `M` is has enough `ETH`, she can fire the following sequence:

$$\begin{aligned}
S \mid \Delta &\xrightarrow{\mathtt{M}:\mathtt{Option.buy()}} && \mathtt{M}[300 : \mathtt{ETH}] \mid \mathtt{AMM}[600 : \mathtt{ETH}, 600 : \mathtt{T}] \mid \mathtt{Option}[20 : \mathtt{ETH}, \cdots] \\
&\xrightarrow{\mathtt{M}:\mathtt{AMM.swap(ETH,300,0)}} && \mathtt{M}[200 : \mathtt{T}] \mid \mathtt{AMM}[900 : \mathtt{ETH}, 400 : \mathtt{T}] \mid \mathtt{Option}[20 : \mathtt{ETH}, \cdots] \\
&\xrightarrow{\mathtt{M}:\mathtt{Option.exercise()}} && \mathtt{M}[20 : \mathtt{ETH}, 200 : \mathtt{T}] \mid \mathtt{AMM}[900 : \mathtt{ETH}, 400 : \mathtt{T}] \mid \mathtt{Option}[0 : \mathtt{ETH}, \cdots] \\
&\xrightarrow{\mathtt{M}:\mathtt{AMM.swap(T,200,0)}} && \mathtt{M}[320 : \mathtt{ETH}] \mid \mathtt{AMM}[600 : \mathtt{ETH}, 600 : \mathtt{T}] \mid \mathtt{Oracle}[0 : \mathtt{ETH}, \cdots]
\end{aligned}$$

where the third transition succeeds because $\mathtt{AMM.getRate}(\mathtt{T}, \mathtt{ETH}) = 9/4 \cdot 10^6 > 2 \cdot 10^6$. This shows that, when `M` can interact with `Option`'s dependency `AMM`, the maximal loss inflictable by `M` to the option contract is:

$$\mathrm{MEV}(S \mid \Delta, \{\mathtt{Option}\}) \;=\; 10 \cdot \$\mathbf{1}_{\mathtt{ETH}}$$

Instead, when `M` is restricted to calling only `Option`, then `M` has no way to remove tokens from the contract in the state S, and so:

$$\mathrm{MEV}_{\{\mathtt{Option}\}}(S \mid \Delta, \{\mathtt{Option}\}) \;=\; 0$$

Therefore, `Option` does not enjoy MEV non-interference with S. Note that a poorer `M` may not have enough `ETH` to produce the short-term volatility in the AMM's exchange rate.

MEV non-interference holds, instead, when using the `PriceOracle` in Listing 1, since no adversary (except the oracle owner itself) can manipulate the exchange rate. Formally, let S be a contract state including the `PriceOracle`. We have that:

- there are no indirect token flows from S to the option contract, and
- the option contract is observationally invariant in S.

Therefore, $S \not\rightsquigarrow \Delta$ follows from Theorem 14(2). By Theorem 17(3), MEV non-interference also holds against wealthy adversaries.

Note that, while the composition `AMM/Option` is correctly classified as *non* 0-composable by [8], the same happens for `PriceOracle/Option` in the states when the adversary (honestly) wins the bet. This seems intuitively incorrect, but technically 0-composability does not hole because in [8] an adversary winning the bet is indistinguishable from extracting MEV.

7.4. Swap proxy

```
contract SwapProxy is IExchange {
  IExchange immutable ex0; IExchange immutable ex1;

  constructor(IExchange ex0_, IExchange ex1_) { ex0=ex0_; ex1=ex1_; }

  function getRate(token ti, token to) public view returns(uint) {
    uint r0 = ex0.getRate(ti,to);
    uint r1 = ex1.getRate(ti,to);
    return (r0 >= r1) ? r0 : r1;
  }
  // assumption: the two exchanges handle the same token types
  function getTokens() public view returns (token[] memory) {
    return ex0.getTokens();
  }

  function swap(token ti, uint x, token to, uint ymin) public returns (uint) {
    ti.receive(x);
    IExchange bex; // the exchange with the maximum exchange rate
    bex = (ex0.getRate(ti,to) >= ex1.getRate(ti,to)) ? ex0 : ex1;
    ti.send(bex,x);
    uint y = bex.swap(ti,x,to,ymin); // reverts if bex outputs < ymin:to
    to.receive(y);
    to.send(msg.sender,y);
    return y;
  }
}
```

Listing 11. A contract that composes two exchanges, swapping with the one with higher rate.

The `SwapProxy` contract in Listing 11 composes two exchanges, providing a `swap` method that forwards the call to one of the underlying exchanges. More precisely, when `swap` is called, it queries the current rates of the two underlying exchanges, and selects the one with the maximum rate to perform the swap. The composed exchanges must implement the `IExchange` interface in Listing 2 (for example, possible instantiations are the `Exchange` and the `AMM` contracts), and must handle the same token types. Note that, in general, `SwapProxy` does not guarantee the optimal swap that one could achieve leveraging the underlying exchanges, for various reasons. First, the queried exchange rate is only the *marginal* rate, i.e. the rate obtained for infinitely small trades. In some exchanges (e.g., in the `AMM`), the actual rate also depends on the input amount: swapping more tokens causes the rate to decrease (diminishing returns). Second, it may happen that none of the underlying exchanges has a sufficient amount of output tokens to cover the input: in that case, one could have to split the trade into two smaller trades.

Since the balance of `SwapProxy` is zero in any blockchain state, no tokens can be extracted, and so its MEV is zero. Therefore, by Theorem 14(1), it enjoys MEV non-interference against any blockchain state — regardless of the specific instantiations of the `IExchange` implemented by the underlying exchanges. MEV non-interference also holds against wealthy adversaries, by Theorem 17(1).

```
contract SwapRouter {
  IExchange immutable ex0; IExchange immutable ex1;

  constructor(IExchange ex0_,IExchange ex1_) { ex0=ex0_; ex1=ex1_; }

  function getTokens() public view returns (address[] memory) {
    address[] memory ts0 = ex0.getTokens();
    address[] memory ts1 = ex1.getTokens();
    address[] memory ts = new address[](ts0.length + ts1.length);
    uint k=0;
    for (uint i=0; i<ts0.length; i++) ts[k++]=ts0[i];
    for (uint i=0; i<ts1.length; i++) ts[k++]=ts1[i];
    return ts;
  }

  function swap(token[] memory path, uint x, uint ymin) public returns (uint) {
    require (path.length >= 2);
    path[0].receive(x);
    uint y = x;
    for (uint i=0; i<path.length-1; i++) {
      IExchange ex = _findExchange(path[i],path[i+1]);
      require (ex!=address(0)); // no exchange for hop
      path[i].send(ex, y);
      y = ex.swap(path[i], path[i+1], y);
      to.receive(y);
    }
    require (y>=ymin);
    path[path.length-1].send(msg.sender, y);
    return y;
  }

  function _findExchange(token a, token b) internal view returns (IExchange) {
    // Find the exchange supporting the swap which has the highest exchange rate
    if (_supports(ex0,a,b)) {
        if (_supports(ex1,a,b) && ex0.getRate(a,b) < ex1.getRate(a,b)) return ex1;
        return ex0;
    }
    if (_supports(ex1,a,b)) return ex1;
    return IExchange(address(0));
  }

  function _supports(IExchange ex, token a, token b) internal view returns (bool) {
    token[] memory ts = ex.getTokens();
    bool hasA = false;
    bool hasB = false;
    for (uint i=0; i<ts.length; i++) {
      if (ts[i]==a) hasA = true;
      if (ts[i]==b) hasB = true;
    }
    return hasA && hasB;
   }
}
```

Listing 12. A contract to route swaps across two exchanges.

7.5. Swap router

The `SwapRouter` contract in Listing 12 composes two exchanges and exposes a single `swap` method that enables users to perform token exchanges along a user-specified path. Each underlying exchange implements the `IExchange` interface and may support swaps among multiple token types, as specified by its `getTokens` method. A swap is defined by a path $\langle \mathtt{T}_0, \mathtt{T}_1, \dots, \mathtt{T}_n \rangle$, indicating a sequence of intermediate token conversions. The router receives an input amount of $\mathtt{T}_0$ from the caller and executes a sequence of swaps $\mathtt{T}_i \to \mathtt{T}_{i+1}$ for $i = 0, \dots, n-1$. For each hop, the router dynamically selects one of the two exchanges that supports both tokens involved in the conversion. If no such exchange exists for some hop, the transaction reverts. The router itself does not expose any pricing; rather, it delegates each hop to the selected underlying exchange, approving the required token transfers. After completing the final hop, the router transfers the resulting tokens to the caller.

Since the balance of `SwapRouter` is zero in any blockchain state, there is nothing to extract, and so its MEV is zero. Therefore, by Theorem 14(1), it enjoys MEV non-interference against any blockchain state — regardless of the implementations of the `IExchange` interface. MEV non-interference also holds against wealthy adversaries, by Theorem 17(1).

```
interface ILoaner {
  function deposit(uint x) external;
  function borrow(uint x) external;
  function repay(uint x) external;
}

contract LP is ILoaner {
  token immutable tB; token immutable tC; // borrowed & collateral
  struct Position { uint credit; uint debt; uint lastAccrual; }
  mapping(address => Position) public pos;
  uint constant BT = 150e4;  // borrowing threshold = 150%
  uint constant LT = 120e4;  // liquidation threshold = 120%
  IPriceOracle oracle; IInterestRate irm;

  constructor(IPriceOracle o,IInterestRate i,token tB_,token tC_) {
    oracle=o; irm=i; tB=tB_; tC=tC_;
    require (tB!=tC && o.getRate(tB,ETH)>0 && o.getRate(tC,ETH)>0);
  }
  function _accrue(address user) internal {
    Position storage p=pos[user];
    if (p.debt==0) { p.lastAccrual=block.number; return; }
    uint interest = irm.getInt(p.debt,p.lastAccrual);
    p.debt += interest; p.lastAccrual=block.number;
  }
  function _isColl(address a) internal view returns (bool) {
    uint vd = pos[a].debt * oracle.getRate(ETH,tB);
    if (vd==0) { return true; } // no debt => collateralized
    uint vc = pos[a].credit * oracle.getRate(ETH,tC);
    return (vc*1e6 >= vd*BT);
  }
  function deposit(uint x) public {
    tC.receive(x);
    pos[msg.sender].credit += x;
  }
  function borrow(uint x) public {
    _accrue(msg.sender);
    tB.send(msg.sender,x);
    pos[msg.sender].debt += x;
    require _isColl(msg.sender);
  }
  function repay(uint x) public {
    _accrue(msg.sender);
    require (pos[msg.sender].debt>=x);
    tB.receive(x);
    pos[msg.sender].debt -= x;
  }
  function liquidate(address borrower) public {
    _accrue(borrower);
    uint vd = (pos[user].debt   * prices.getPrice(tB))/1e6;
    uint vc = (pos[user].credit * prices.getPrice(tC))/1e6;
    require (vc*1e6 < vd*LT);
    tB.receive(pos[borrower].debt);
    tC.send(msg.sender,pos[borrower].credit);
    pos[borrower].debt = 0;
    pos[borrower].credit = 0; // simplified full liquidation
  }
}
```

Listing 13. A lending pool contract.

7.6. Lending pool

The contract LP in Listing 13 implements a simplified lending protocol that allows users to deposit a collateral token T_c and borrow a distinct token T_b. The protocol relies on two external contract dependencies:

- a *price oracle*, used to compute the value of collateral and outstanding debt;
- an *interest rate model*, used to compute interests on loans (Listing 14).

```
interface IInterestRate {
  // Returns interest accrued since `last` block on `debt`
  function getInt(uint debt, uint last) external view returns(uint);
}

// Simple linear interest per block
contract LinearInterestRate is IInterestRate {
  uint public immutable ratePerBlock; // e.g., 1e16 = 1% per block

  constructor(uint r) { ratePerBlock = r; }

  function getInt(uint debt, uint last) public view returns (uint) {
    if (debt==0 || block.number<=last) return 0;
    uint blocks = block.number - last;
    return (debt * ratePerBlock * blocks) / 1e18;
  }
}
```

Listing 14. A simple Interest Rate Model.

Our LP contract features the following methods:

- `deposit` allows anyone to deposit tokens $\mathtt{T}_c$ in the pool. These tokens can be used as collateral to borrow tokens $\mathtt{T}_b$. For simplicity, in Listing 13 we omit the method to redeem deposited $\mathtt{T}_c$ tokens.
- `borrow` allows any user with sufficient collateral to withdraw tokens $\mathtt{T}_b$ from the pool. The actual condition is defined in the method `_isColl`: more precisely, borrowing is possible when the value of the user's collateral is at least BT times the value of their debt (after the borrow). In our example, we choose a borrowing threshold $\mathtt{BT} = 150\%$. Note that, once tokens are borrowed, the value of the debt will automatically increase over time following the interest rate model.
- `repay` allows a borrower to return part of their debt.
- `liquidate` allows anyone to repay the debt of an under-collateralized borrower, thus gaining the ownership of the collateral. We consider a borrower to be under-collateralized when the value of their collateral is below LT times the value of their debt. In our example, we choose a liquidation threshold $\mathtt{LT} = 120\%$. Note that a borrower can become under-collateralized for various reasons, such as the accrual of interest on the debt and price fluctuations (e.g., the price of the collateral token $\mathtt{T}_c$ drops). For simplicity, the LP contract in Listing 13 implements a *full liquidation* mechanism, in which the liquidator repays the entire outstanding debt and receives the borrower's entire collateral balance.

Lending protocols of this kind are notoriously subject to MEV [14]. For example, an adversary with the power of causing sufficiently large price fluctuations can cause a borrower's collateral to fall below the threshold, and then trigger liquidation by repaying the borrower's debt and receiving its collateral in exchange. Under suitable conditions on prices, this can cause a loss of value to the LP contract.

We now study the composition of a lending pool contract LP with a SwapRouter serving as a price oracle. The router is based on two AMMs as underlying exchanges. We will show that this composition does *not* enjoy MEV non-interference. Assume that the borrowed and collateral tokens are, respectively, $\mathtt{T}_b$ and $\mathtt{T}_c$, with external prices $\$\mathbf{1}_{\mathtt{T}_b} = 2$ and $\$\mathbf{1}_{\mathtt{T}_c} = 1$. Let $\mathcal{M} = \{\mathtt{M}\}$, and consider the following state S and an LP in Δ where A has deposited $200 : \mathtt{T}_c$ and borrowed $60 : \mathtt{T}_b$:

$$S = \mathtt{A}[0 : \mathtt{T}_c, 60 : \mathtt{T}_b] \mid \mathtt{M}[100 : \mathtt{T}_c, 60 : \mathtt{T}_b] \mid \mathtt{AMM}_1[50 : \mathtt{T}_b, 100 : \mathtt{ETH}] \mid \mathtt{AMM}_2[100 : \mathtt{T}_c, 100 : \mathtt{ETH}] \mid \mathtt{SwapRouter}[\cdots]$$

$$\Delta = \mathtt{LP}[40 : \mathtt{T}_b, 200 : \mathtt{T}_c, \mathtt{oracle} = \mathtt{SwapRouter}, \mathtt{pos} = \{\mathtt{A} \mapsto (200, 60, 0)\}, \ldots]$$

In the state $S \mid \Delta$, we have that:

- the $\mathtt{ETH}/\mathtt{T}_b$ rate is 2, and so the debt value of A is $\mathtt{vd} = 60 \cdot 2 = 120$;
- the $\mathtt{ETH}/\mathtt{T}_c$ rate is 1, and so the collateral of A is $\mathtt{vc} = 200 \cdot 1 = 200$;
- A is not liquidatable, since $\mathtt{vc} = 200 \geq \mathtt{vd} \cdot 120\% = 144$.

Therefore, A is not liquidatable in $S \mid \Delta$: this implies that when M is restricted to call the contracts in Δ, they cannot extract value from the LP. Instead, if M can call any contract in S, they can manipulate the prices in order to make A under-collateralized, and then liquidate A — causing the extraction of value from the LP. The attack sequence $\vec{\mathscr{X}}$ is the following:

1. $\mathtt{M} : \mathtt{AMM}_2.\mathtt{swap}(\mathtt{T}_c, 100, \mathtt{ETH}, 0)$. This changes the AMM state to:

   $$\mathtt{AMM}_2[200 : \mathtt{T}_c, 50 : \mathtt{ETH}]$$

   In this state, the price of $\mathtt{T}_c$ has *decreased* to 0.25, while the price of $\mathtt{T}_b$ and A's debt value remain unchanged, so the collateral value of A is $200 \cdot 0.25 = 50$. Therefore, A is liquidatable, since $\mathtt{vc} = 50 < \mathtt{vd} \cdot 120\% = 144$.
2. $\mathtt{M} : \mathtt{LP}.\mathtt{liquidate}(\mathtt{A})$. The effect of this liquidation is that M repays the full debt $60 : \mathtt{T}_b$ of A and receives her full collateral $200 : \mathtt{T}_c$.
3. $\mathtt{M} : \mathtt{AMM}_2.\mathtt{swap}(\mathtt{ETH}, 50, \mathtt{T}_c, 0)$. The adversary restores the AMM state and the exchange rate $\mathtt{ETH}/\mathtt{T}_c$ as they were in S.

The state after the attack is the following:

$$S' = \mathtt{A}[0 : \mathtt{T}_c, 60 : \mathtt{T}_b] \mid \mathtt{M}[300 : \mathtt{T}_c, 0 : \mathtt{T}_b] \mid \mathtt{AMM}_1[50 : \mathtt{T}_b, 100 : \mathtt{ETH}] \mid \mathtt{AMM}_2[100 : \mathtt{T}_c, 100 : \mathtt{ETH}] \mid \mathtt{SwapRouter}[\cdots]$$
$$\Delta' = \mathtt{LP}[100 : \mathtt{T}_b, 0 : \mathtt{T}_c, \mathtt{pos} = \{\mathtt{A} \mapsto (0,0,0)\}, \ldots]$$

Summing up, the loss of the LP due to this attack sequence is:

$$\begin{aligned} -\gamma_{\{\mathtt{LP}\}}(S, \vec{\mathcal{X}}) &= \$_{\{\mathtt{LP}\}}(S) - \$_{\{\mathtt{LP}\}}(S') \\ &= (40 \cdot \$\mathbf{1}_{\mathtt{T}_b} + 200 \cdot \$\mathbf{1}_{\mathtt{T}_c}) - (100 \cdot \$\mathbf{1}_{\mathtt{T}_b} + 0 \cdot \$\mathbf{1}_{\mathtt{T}_c}) \\ &= 280 - 200 > 0 \end{aligned}$$

This implies that the unrestricted MEV is strictly positive, and consequently it is different from the restricted MEV — which is zero. Therefore, $S \nrightsquigarrow \Delta$.

MEV non-interference can be recovered by replacing the AMM with a price oracle that cannot be manipulated by the adversary. Additionally, we must also fix the instance of IInterestRate to prevent the adversary from influencing the behaviour of LP. For instance, choosing PriceOracle from Listing 1 and LinearInterestRate from Listing 14 makes the lending pool observationally invariant w.r.t. adversarial moves, and so Theorem 17(3) gives non-interference w.r.t. wealthy adversaries. Establishing MEV non-interference w.r.t. "poor" adversaries through Theorem 17(3) would additionally require the absence of token flows from the system to the LP, which is not guaranteed in our composition.

7.7. Lending pool arbitrage

```
contract LPArbitrage {
  ILoaner   immutable lp;  // borrow t0 (with collateral tc)
  IExchange immutable ex0; // swap t0 -> t1
  IExchange immutable ex1; // swap t1 -> t0

  constructor(ILoaner lp_,IExchange ex0_,IExchange ex1_,token tc_,uint n) {
    lp=lp_; ex0=ex0_; ex1=ex1_; tc=tc_;
    tc.receive(n); tc.send(lp,n); lp.deposit(n); // collateral
  }

  function arbitrage(uint x, token t0, token t1) public {
    require (t0!=tc && t1!=tc);
    lp.borrow(x);                  // borrow x:t0
    t0.send(ex0,x);
    uint y1 = ex0.swap(t0,x,t1,0); // sell t0, buy t1
    t1.send(ex1,y1);
    ex1.swap(t1,y1,t0,0);          // sell t1, buy t0
    t0.send(lp,x);
    lp.repay(x);                   // repay x:t0
    uint d=t0.balanceOf(this);
    require (d>0);                 // sender's gain must be positive
    t0.send(msg.sender,d);
  }
}
```

Listing 15. Arbitrage with a Lending Pool.

Arbitrage in DeFi refers to a transaction (or sequence of transactions) that yields a profit by exploiting price discrepancies for the same token pair across different exchanges. This situation typically arises when two exchanges quote different prices for the same token pair, due for instance to independent liquidity pools, delayed oracle updates, or heterogeneous pricing mechanisms. To execute the arbitrage, the arbitrageur must first obtain a sufficient amount of liquidity to perform the initial swap. This liquidity can be sourced in different ways: in this section, we illustrate arbitrage funded through a lending pool, while in the next section we consider the use of a flash loan.

The LPArbitrage contract in Listing 15 composes two exchanges and a Lending Pool to provide a zero-risk arbitrage. To this purpose, the function arbitrage(x) performs atomically the following sequence of transactions:

1. borrow $x : \mathtt{T}_0$;
2. swap $x : \mathtt{T}_0$ for a certain amount $x_1 : \mathtt{T}_1$;
3. swap $x_1 : \mathtt{T}_1$ for a certain amount $y : \mathtt{T}_0$;
4. repay the loan $x : \mathtt{T}_0$;
5. transfer the gained units $d = y - x > 0$ of $\mathtt{T}_0$ to the caller.

Since the balance of LPArbitrage is zero in any blockchain state, there is nothing to extract, and so its MEV is zero. Therefore, by Theorem 14(1), it enjoys MEV non-interference against any blockchain state — regardless of the implementations of the ILoaner and IExchange interfaces. MEV non-interference also holds against wealthy adversaries, by Theorem 17(1).

7.8. Flash loan arbitrage

```
interface IFlashLoaner {
  function flashLoan(uint amount, token t) external;
}

interface IFlashBorrower {
  function exec(uint amount) external;
}

contract FlashLP is LP, IFlashLoaner {
  uint public immutable fee;

  constructor(IPriceOracle o,IInterestRate i,token tB_,token tC_,uint f) LP(o,i,tB_,tC_) {
      fee = f; }

  function flashLoan(uint amt, token t) public {
    require (t==tB);
    uint oldBal = t.balanceOf(address(this));
    t.send(msg.sender,amt);
    IFlashBorrower(msg.sender).exec(amt);
    require (t.balanceOf(address(this)) >= oldBal+fee);
  }
}
```

Listing 16. A flash loan contract.

```
contract FlashLoanArbitrage is IFlashBorrower {
  IFlashLoaner immutable lp; // to borrow t0 (without collateral)
  IExchange immutable ex0;   // to swap t0 -> t1
  IExchange immutable ex1;   // to swap t1 -> t0
  token immutable t0; token immutable t1;

  constructor(IFlashLoaner lp_,IExchange ex0_,IExchange ex1_) {
    lp=lp_; ex0=ex0_; ex1=ex1_;
  }

  function arbitrage(uint x, token t0_, token t1_) public {
    t0=t0_; t1=t1_; lp.flashLoan(x,t0);
  }

  function exec(uint amt) public {
    t0.receive(amt);                  // from lp
    t0.send(ex0,x);
    uint x1 = ex0.swap(t0,x,t1,0);    // sell t0, buy t1
    t1.send(ex1,x1);
    ex1.swap(t1,x1,t0,0);             // sell t1, buy t0
    t0.send(msg.sender,amt+lp.fee()); // repay flash loan
    uint d=t0.balanceOf(this)
    require (d>0);                    // sender's gain must be positive
    t0.send(tx.origin,d);
  }
}
```

Listing 17. Arbitrage with a flash loan.

The contract `LParbitrage` seen before requires the arbitrageur to have a collateral in `LP` in order to perform the `borrow` action. Current lending protocols, such as e.g. Aave [22], also feature a `flashLoan` method that allows users to borrow tokens without providing a collateral, at the condition that the loan (plus a fee) is repaid in the *same* transaction.

We show in Listing 16 a lending protocol exposing flash loans, and in Listing 17 an arbitrage contract relying on a flash loan. Note that the dependencies between `FlashLoanArbitrage` and `FlashLoan` are circular, since the method `arbitrage` calls `flashLoan`, which calls back `exec`. This breaks well-formedness of blockchain states (see Section 2). However, well-formedness is *not* required by Theorems 14 and 17, which establish, in their first item, MEV non-interference whenever the newly deployed contracts have zero MEV. This is the case for `FlashLoanArbitrage`: regardless of the dependencies, the contract balance is always zero, so there is nothing to extract.

8. Related work

A primary term of comparison for our MEV non-interference is the ε-composability notion of [8]:

$$\Delta \text{ is } \varepsilon\text{-composable in } S \quad \text{iff} \quad \mathrm{MEV}(S \mid \Delta) \leq (1+\varepsilon)\,\mathrm{MEV}(S)$$

The key conceptual difference between the two notions is that ε-composability is a global property of the *entire* blockchain state, whereas MEV non-interference has a more local nature, as it only requires inspecting the newly deployed contracts and their dependencies. Indeed, the "only if" direction of the implication in Theorems 15 and 18 states that, to establish the MEV non-interference of Δ in state $\Sigma \mid \Gamma$, we can neglect the part of the state Γ of the non-dependencies of Δ, under suitable conditions (e.g., $deps(\Delta)$ is sender-agnostic). Besides that, the two notions are complementary, in that they have different purposes: ε-composability asks whether adding new contracts increases the global MEV of the system, while MEV non-interference asks whether the adversary can inflict a loss to new contracts by exploiting their dependencies.

Technically, ε-composability and MEV non-interference (in both "rich" and "poor" versions) are incomparable notions. To show that MEV non-interference does not imply ε-composability, assume that the newly deployed contracts Δ contains exactly the `Airdrop` contract from Example 12, with a wallet of $n : \mathtt{T}$. In $S \mid \Delta$, the adversary can extract $n : \mathtt{T}$ from `Airdrop`, and possibly use these tokens to extract more MEV from S. Therefore, we have $\mathrm{MEV}(S \mid \Delta) \geq n \cdot \$\mathbf{1}_{\mathtt{T}} + \mathrm{MEV}(S)$, which implies that, if n is large enough, the airdrop is *not* ε-composable with S. Example 12 shows, instead, that MEV non-interference holds.

Example 18 below shows that the converse implication fails as well.

Example 18. We craft 0-composable contracts which violate MEV non-interference. For this, we define Γ and Δ such that both contracts expose identical and *mutually exclusive* MEV: one can extract MEV from either contract, but not from both. The choice of extracting MEV from Γ or from Δ is done by calling a suitable method of Γ. Only after the choice is made the MEV is exposed, and the MEV from the other contract is permanently disabled. Since the two MEVs are mutually exclusive, we obtain $\mathrm{MEV}(\cdots \mid \Gamma) = \mathrm{MEV}(\cdots \mid \Gamma \mid \Delta)$, hence 0-composability. Non-interference fails because extracting MEV from Δ requires calling a method of Γ.

More formally, consider the contracts in Listing 18, and let $\Gamma = \mathtt{C1}[1 : \mathtt{T}, \mathtt{n} = 0]$, $S = \mathtt{M}[0 : \mathtt{T}] \mid \Gamma$, and $\Delta = \mathtt{C2}[1 : \mathtt{T}]$. To study ε-composability, we compare the (global) MEV of S and $S \mid \Delta$. We have that $\mathrm{MEV}(S) = 1 \cdot \$\mathbf{1}_{\mathtt{T}}$ by the sequence `M : C1.f1()`, and $\mathrm{MEV}(S \mid \Delta) = 1 \cdot \$\mathbf{1}_{\mathtt{T}}$ by the sequence `M : C1.f2() M : C2.g()` (or, alternatively, by the sequence `M : C1.f1()`, which provides the same MEV). Therefore, the two contracts are 0-composable.

For MEV non-interference, we have that $\mathrm{MEV}(S \mid \Delta, \{\mathtt{C2}\}) = 1 \cdot \$\mathbf{1}_{\mathtt{T}}$ by the sequence `M : C1.f2() M : C2.g()`, while $\mathrm{MEV}_{\{\mathtt{C2}\}}(S \mid \Delta, \{\mathtt{C2}\}) = 0$, since the only callable method, i.e. `g()`, always fails. Therefore, $S \nrightsquigarrow \Delta$, i.e. Δ is not composable with S according to our notion. ◇

Another difference between MEV non-interference and ε-composability is highlighted by the first item of Theorems 14 and 17, which state that if the newly deployed contracts have zero MEV, then MEV non-interference holds. This is not, instead, a sufficient condition for ε-composability: a contract may not be 0-composable in S when $\mathcal{M}$ does not have the tokens needed to extract MEV in S, but becomes able to do so after exchanging tokens between S and Δ, provided that this preserves Δ's wealth (see Example 19).

```
contract C1 {
  uint public n;
  constructor() { T.receive(1); n=0; }
  function f1() { require (n==0); n=1; T.send(msg.sender,1); }
  function f2() { require (n==0); n=2; }
}
contract C2 {
  constructor() { T.receive(1); }
  function g() { require C1.n()==2; T.send(msg.sender,1); }
}
```

Listing 18. Babel et al.' composability does not imply MEV non-interference.

Example 19. Using the states S and Δ in Example 7, we show that a contract with zero MEV may not be 0-composable with the context. Note that any action on `Exchange2` does not alter its wealth, since the total number of tokens held in that contract is preserved (because their internal prices given by `PriceOracle2` are equal), and they have the same (external) price. Therefore, $\mathrm{MEV}(S \mid \Delta, \{\mathtt{Exchange2}\}) = 0$, and so condition 1 of Theorem 14 ensures that S is MEV non-interfering with Δ.

Instead, Δ is *not* 0-composable with S: indeed, $\mathrm{MEV}(S) = 0 \neq \mathrm{MEV}(S \mid \Delta) = 1$, since `M` can first swap her $1 : \mathtt{T}_2$ for $1 : \mathtt{T}_1$ using `Exchange2`, and then swap $1 : \mathtt{T}_1$ for $2 : \mathtt{T}_2$ using `Exchange1`, with an overall gain of $1 : \mathtt{T}_2$. This is coherent with the different interpretations of composability: for [8], adding a new contract must preserve the global MEV, while our notion requires to preserve the MEV of the victim contracts. ◇

The work [23] applies classical program non-interference notions to the setting of smart contracts. There, smart contracts are specified in a general-purpose concurrent imperative language. Unlike in our approach, where token holdings and exchanges are modelled implicitly in the contract semantics, in [23] they must be explicitly programmed as assignments to suitable variables. Therefore, measuring the MEV of a contract is reduced to estimating the change of such variables. Similarly, establishing non-interference amounts to detecting whether some high-level variables (e.g., the rate of an exchange contract) can affect low-level variables (e.g., the wallet of a contract using that exchange). Therefore, this approach requires the programmer to identify the high-level and the low-level variables, and to manually downgrade some variables from high to low to explicitly admit some flows (e.g., the oracle owner can update the exchange rate). Our approach, instead, does not require the programmer to annotate the contract code: roughly, the "low" variable is the MEV extractable from the victim contracts, and the "high" variables are the methods callable from the adversary.

This observation reveals a key difference between our approach and [23]: MEV non-interference may hold also when the adversary can affect the victim contracts' wallet, provided that their loss is preserved, while in [23] any flow from the variables touched by the adversary to the wallet is considered harmful. For example, consider a variant of the `Airdrop` contract that allows its users to withdraw either a token $\mathtt{T}_1$ or another token $\mathtt{T}_2$, where both tokens have the same external price. This `Airdrop` contract can only be used once, and internally chooses which token to send by querying another contract `PriceOracle`. According to our notion of MEV non-interference, there is no interference between `PriceOracle` and the `Airdrop`, since the extracted token will have the same price, hence generating the same MEV. Instead, the non-interference notion in [23] can observe that `PriceOracle` can interfere with `Airdrop`, affecting its behavior.

More than with MEV non-interference, the notion of non-interference in [23] seems related to our observational invariance (Section 3.4). Informally, this notion requires that no action performed by the adversary on S can influence what the contracts in Δ are able to observe from S. Conceptually, this resembles a classic non-interference property, where high-level variables (i.e., the methods of the contracts in S) are prevented from affecting low-level variables (i.e., the observations that Δ can make on S).

The work [24] introduces a *quantitative* versions MEV non-interference, which enables to measure the effect of MEV attacks on the victim contract. More specifically, given a set of victim contracts Δ and a system S, the value $\mathcal{I}(S \rightsquigarrow \Delta) \in [0, 1]$ measures the increase of loss (proportionally to Δ's wealth) that adversaries can inflict to Δ by manipulating the context S. In particular, a value $\mathcal{I}(S \rightsquigarrow \Delta) = 0$ means that no such increase is possible: this is coherent with our notion, i.e. $\mathcal{I}(S \rightsquigarrow \Delta) = 0$ holds iff $S \not\rightsquigarrow \Delta$. Instead, a larger (positive) value of $\mathcal{I}(S \rightsquigarrow \Delta)$ means that a larger fraction of Δ's wealth can be extracted by the adversary. The quantitative nature of this notion makes it possible to have more fine-grained results about the effect of adding new contracts to the system. In particular, $\mathcal{I}(S \rightsquigarrow \Delta)$ increases when S is extended with contracts that are not in the dependencies of Δ, i.e. $\mathcal{I}(S \rightsquigarrow \Delta) \leq \mathcal{I}(S \mid \Gamma \rightsquigarrow \Delta)$. By contrast, our state narrowing results (Theorems 15 and 18) can only elaborate on conditions about Γ that preserve MEV non-interference. A potential advantage of a quantitative notion *vs.* a qualitative one is to understand the effect of different system parameters on the MEV increase. The work [24] exemplifies this by measuring the degree of interference in compositions akin to our `AMM/Option` and `AMM/LP` use cases in Section 7. However, establishing non-trivial interference bounds in complex contract compositions seems quite more challenging than establishing our MEV non-interference.

A preliminary version of this work appeared in [25]. The present article substantially extends that work, both in its theoretical development and in its analysis of DeFi compositions. On the theoretical side, we refine the conditions under which state narrowings/widenings preserve MEV and MEV non-interference. In particular, while [25] established local reasoning principles only for unbounded-wealth adversaries, the current work extends these results to bounded-wealth adversaries. This yields preservation theorems for both MEV and $\not\rightsquigarrow$ under suitable conditions (Theorems 5 and 15, respectively), thereby providing a unified treatment of the two adversarial models. On the practical side, the current version broadens the analysis of DeFi compositions. In [25], the swap proxy, swap router, and arbitrage protocols were studied only in conjunction with AMM-based exchanges. Here, we generalise the analysis to arbitrary exchanges implementing the required interfaces (Table 2). In addition, the present version streamlines and generalises several technical developments, and provides complete proofs for all results.

9. Limitations

We discuss below some simplifications of our approach, which overall may affect its applicability to real-world smart contract platforms.

Token prices. Our definition of the price of a token $\mathtt{T}$ as $\$\mathbf{1}_{\mathtt{T}}$ treats token prices as exogenous parameters, in that such prices do not depend on the blockchain state. As a consequence, our composability notions are parametric in the chosen price assignment, and a composition classified as MEV non-interfering under one assignment need not remain so under another.

One possible way to eliminate this parameter would be to internalize prices in the blockchain state, for instance by designating certain contracts as price oracles (e.g., contracts exposing the interface `IPriceOracle`). This would make prices endogenous, but it would also make the resulting security notions depend more heavily on the particular oracle contracts included in the state (an on their transitions). In this sense, the dependency on external prices would not really disappear, but would instead be shifted to a dependency on the oracle mechanisms used to determine them.

A more robust alternative is to rely on stricter, price-independent notions of secure composability that avoid collapsing heterogeneous token movements into a single MEV scalar. Our observational invariance provides one such notion: it is independent of token prices by definition (Definition 5), and it implies MEV non-interference under the sufficient conditions established in Theorems 14 and 17. Thus, observational invariance can be seen as a conservative, price-agnostic criterion for secure composability, particularly useful when no canonical or stable price assignment is available.

Contract model. Our model does not allow contracts to inspect the blockchain state directly, for example by querying token balances of other accounts or the storage of other contracts (Section 2). This restriction does not reduce expressiveness, since such information can be exposed through simple mechanisms such as getter methods or explicit token inputs. It nevertheless marks a difference from EVM-based platforms such as Ethereum, where contracts can access parts of the blockchain state directly. The contract language we have used throughout the paper to write examples is a simplification of Solidity that respects the constraints of our model (with the exception of the native handling of tokens). We expect that the analysis of our examples (e.g., the classification of DeFi compositions in Section 7) can be directly lifted to Ethereum, by suitably translating the contracts from our language into Solidity.

Another point of divergence is our well-formedness condition, which is assumed by many of our results, although some key results — including the sufficient conditions for MEV non-interference, Theorems 14 and 17 — do not require it. Some of our results, instead, rely on the assumption that dependencies are backward-closed, i.e. a contract cannot invoke a contract deployed after itself. For example, the monotonicity result of Lemma 2(5), which plays a key role in establishing the front-running resistance of $\not\rightsquigarrow^{\infty}$, fails without this assumption (see Example 6). Backward-closure is a simplification with respect to Ethereum, where dependencies may be dynamic and even cyclic. In particular, contracts can interact through reentrant calls, enabling attacks that exploit existing contracts to extract additional MEV, as exemplified by the DAO attack [26]. A possible research direction is to relax these assumptions by localizing them to the contracts under analysis. Rather than requiring backward-closure throughout the entire blockchain state, one could investigate whether our results continue to hold under assumptions that only constrain the newly deployed contracts Δ and their dependency graph. This would leave the surrounding state S unconstrained, allowing arbitrary pre-existing dependencies and interactions among already deployed contracts. With this relaxation, the rest of the blockchain state would not be constrained, making our results applicable to a wider class of contracts.

Freezing attacks. Our notion of MEV measures the loss of a contract as the value of the tokens that adversaries can remove from it. In practice, adversaries could harm contracts also by *freezing* tokens without actually extracting them from the contract, as in the infamous Parity Wallet attack [27]. Refining local MEV to take these attacks into account could be done by adapting notions such as solvency [28], *enabledness* [29] and liquidity [30–32].

UTXO-based blockchains. Our study focusses on smart contract platforms following the account-based accounting model, the one used in Ethereum and several other blockchains [33]. Extending our results to other accounting models, such as the UTXO model, seems challenging. Indeed, our MEV non-interference analyses contract-to-contract interactions such as (cross-) contract calls and token flows. In pure UTXO-based platforms such as Bitcoin, contract calls are not implementable [34]. In extended UTXO-based models, such as the one implemented by Cardano [35], some forms of contract calls are possible, even though they require more indirect mechanisms such as message-passing interactions encoded within token minting policies [36]. Because of this additional complexity, contract calls are not commonly used in UTXO-based smart contracts, which tend to be more isolated and avoid dependencies, compared to contracts in the account-based model [37]. Without contract calls, interactions between contracts are essentially reduced to token flows (Section 2): for example, a contract can allow a user to withdraw a token which is then sent to another contract to trigger a certain behaviour. In this scenario, MEV non-interference becomes a much more limited notion, and less interesting to observe.

Transaction and gas fees. Our definition of MEV does not take transaction and gas fees into account. This choice is in line with common abstractions adopted in literature [8,14,38]. In principle, our definitions can be adapted to address these additional costs. Doing so would restrict the adversary to transaction sequences whose total execution cost is within the adversary's budget, ruling out executions that the adversary cannot afford. In such scenarios, both MEV and MEV non-interference would then depend not only on the wealth of the adversary, but also on the actual fee mechanism of the smart contract platform at hand, resulting in a more complex theory. We expect our results to hold even in the presence of fees, under suitable assumptions on the fee mechanism.

When dealing with wealthy adversaries, instead, fees are substantially immaterial: since their costs are paid by the adversary itself, any transactions sequence executable in the fee-lees model is also such in the fee-aware one, and *vice-versa*. Furthermore, since our MEV notion observes the contract loss (instead of the adversarial gain, as in [8]), the additional costs the adversary must cover do not affect MEV. Therefore, for wealthy adversaries, the notions of MEV and MEV non-interference are not affected by fees.

Mempool. Our model implicitly assumes an empty mempool. In certain scenarios, access to pending transactions in the mempool increases the adversary's opportunities for MEV extraction. A notable example is the *sandwich attacks* on AMMs [39], where the adversary observes a user's trade in the mempool and strategically places transactions before and after it, causing the user's trade to execute under less favourable conditions, while generating a profit for the adversary. Our framework abstracts away from the mempool because our primary goal is to study secure composability, namely the additional MEV opportunities introduced by newly deployed contracts rather than those arising from pending transactions. While incorporating the mempool into our definitions would be consistent with the principle of analysing security notions under increasingly stronger adversaries (as we did when considering wealthy adversaries), doing so would considerably increase the complexity of an already technically demanding theory, while providing limited additional insight into the composability issues that are the focus of this work. Making our MEV and MEV non-interference depend on the mempool seems feasible, e.g. following the definitions in [14].

10. Conclusions and discussion

We have proposed MEV non-interference, a new security notion for DeFi composition which ensures that adversaries cannot inflict economic harm on compound contracts by exploiting their dependencies. In particular, we have studied sufficient conditions for MEV non-interference (Theorems 14 and 17), and rules to enable its modular verification (Theorems 15 and 18). We have shown that these rules allow to correctly classify the composability of several common DeFi protocols (Table 2).

Table 3
Some non-properties of $\not\rightarrow^{\infty}$ (the hypothesis does not imply the thesis).

Hypothesis	Thesis	Counterexample
$\Gamma \not\rightarrow^{\infty} \Delta$	$\Gamma \not\rightarrow^{\infty} \Delta \mid \Delta'$ $\Gamma \not\rightarrow^{\infty} \Delta' \mid \Delta$	Example 23 Example 23
$\Gamma \not\rightarrow^{\infty} \Delta \mid \Delta'$ $\Gamma \not\rightarrow^{\infty} \Delta' \mid \Delta$	$\Gamma \not\rightarrow^{\infty} \Delta$	Example 24 Example 24
$\Gamma \not\rightarrow^{\infty} \Delta_1 \wedge \Gamma \not\rightarrow^{\infty} \Delta_2$ $\Gamma \not\rightarrow^{\infty} \Delta_1 \wedge \Gamma \mid \Delta_1 \not\rightarrow^{\infty} \Delta_2$	$\Gamma \not\rightarrow^{\infty} \Delta_1 \mid \Delta_2$	Example 25 Example 25

Alternative definitions. A relevant question is whether simpler notions of composability would achieve the same effect as our Definition 8. For instance, one might be tempted to regard two contracts $\mathtt{C}$ and $\mathtt{C}'$ composable whenever the (global) MEV of their composition is equal to the sum of the two individual MEVs, thus obtaining a property which could be written along the line of

$$\mathrm{MEV}(\mathtt{C}[\cdot] \mid \mathtt{C}'[\cdot]) = \mathrm{MEV}(\mathtt{C}[\cdot]) + \mathrm{MEV}(\mathtt{C}'[\cdot]) \tag{12}$$

While temptingly simple, this notion has several issues. First, it does not consider the dependencies of the two contracts, which must be part of the blockchain state. E.g., when both contracts depend on a third contract $\mathtt{D}$, we could amend (12) as:

$$\mathrm{MEV}(\mathtt{D}[\cdot] \mid \mathtt{C}[\cdot] \mid \mathtt{C}'[\cdot]) = \mathrm{MEV}(\mathtt{D}[\cdot] \mid \mathtt{C}[\cdot]) + \mathrm{MEV}(\mathtt{D}[\cdot] \mid \mathtt{C}'[\cdot]) \tag{13}$$

However, this equation is almost always false, since in the RHS the MEV extractable from $\mathtt{D}$ is counted *twice*. Even when $\mathtt{D}$ has no MEV, it can still act as a shared state between $\mathtt{C}$ and $\mathtt{C}'$, e.g. making it possible to extract MEV from only one of them (but not both). This, again, can be used to falsify (13). Furthermore, (13) does not mention the adversary wallet. Using the same wallet in each $\mathrm{MEV}(\cdot)$ would duplicate the adversary wealth in the RHS, leading to similar double-counting issues as those discussed previously for $\mathtt{D}$.

Another alternative definition of MEV could be the following:

$$\mathrm{MEV}_D(S, C) = \max \left\{ -\gamma_C(S, \vec{\mathcal{X}}) - \gamma_{\mathbb{A}_u \setminus \mathcal{M}}(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^* \right\} \tag{14}$$

Compared to (3), the MEV in (14) subtracts, from the loss of the victim contracts C, the gain of the addresses $\mathbb{A}_u \setminus \mathcal{M}$ not controlled by the adversary. The intuition is that if a transaction fired by the adversary has the effect of moving tokens from a victim contract to a "legit" user (e.g., one that should have received the tokens anyway), then it should not count as MEV. The problem with this alternative definition is that "legit" addresses seem hard to characterise: for example, if the adversary manages to move tokens from the victim contracts to a sink contract from where no tokens can be extracted, or to a randomly chosen address, then it would be make more sense to count these tokens as a loss, unlike (14).

Pitfalls. MEV non-interference enjoys some interesting structural properties: notably, Theorems 15 and 18 show that, under certain conditions, non-interference is preserved when extending or restricting the state. At the same time, a number of seemingly plausible structural properties do not hold in general. Table 3 summarises several such non-properties of $\not\rightarrow^{\infty}$, which — by Lemma 16 — also fail to hold for $\not\rightarrow$.

The first two sections show that, in general, adding or removing contracts from Δ does not preserve MEV non-interference. While it is intuitively obvious that adding contracts breaks non-interference, the fact that removing them also breaks it is somehow surprising. The last section of Table 3 is even more surprising: proving that Γ does not interfere with both Δ_1 and Δ_2 is not enough to guarantee that Γ does not interfere with the composition $\Delta_1 \mid \Delta_2$. Since the counterexamples needed to refute these conjectures are quite technical, we relegate them to D.

Adversary model. Our definition of MEV is parameterised by a set of addresses $\mathcal{M}$ controlled by the adversary. In principle, different choices of $\mathcal{M}$ may yield different MEV values and, consequently, different non-interference classifications for the same contract composition. In practice, however, any sufficiently large set of addresses that does not already occur in the contracts under analysis (i.e., neither in their storage nor as hard-coded constants in their code) would yield the same MEV. It would therefore be possible to reformulate our notions to as to eliminate any dependence on the particular choice for $\mathcal{M}$, albeit at the cost of an increased complexity of the definitions and statements.

A similar approach has been adopted in [14] in the context of the classical interpretation of MEV as maximal adversarial gain. There, the authors introduce the notion of *universal MEV*, defined as the maximum gain that can be extracted over all cofinite sets of addresses $\mathcal{M}$, after suitably redistributing tokens among the wallets controlled by $\mathcal{M}$. Besides abstracting from the actual identity of the adversary (i.e., from the choice of the set $\mathcal{M}$), this notion also abstracts from their wealth, in this second respect being conceptually similar to our MEV^{∞} (Definition 7). The two notions differ, however, in the assumptions they make about the adversary's available tokens. In universal MEV, the adversary's capital is ultimately bounded by the total amount of tokens already occurring in the blockchain state (since the definition just redistributes that to $\mathcal{M}$). In contrast, our definition of MEV^{∞} assumes that the adversary has access to whatever capital is required to execute the attack.

Toolchain. Turning our theory into a practical framework would require a software toolchain that, given a set of dependencies and a target victim contract (written, e.g., in Solidity), detects whether the victim enjoys MEV non-interference against the dependencies. A first obstacle is that MEV non-interference is undecidable: this is not surprising, since the underlying contract language is Turing-complete. Therefore, suitable safe and computable approximations of MEV non-interference are in order. Theorems 14 and 17 are a first step towards that goal, since they provide sufficient conditions for MEV non-interference. In particular, their first condition, i.e. that the victim contracts have zero MEV, is undecidable in general, but in some cases it can be safely approximated, e.g. when the contract cannot receive tokens, or all the tokens it sends go to its owner. A similar observation holds for the absence of token flows and observational invariance: both notions are undecidable in general, but can be approximated. For instance, if the state of the dependencies of the victim contract can only by modified by their respective owners (e.g., as in Example 9), then observational invariants can be easily established.

CRediT authorship contribution statement

Massimo Bartoletti: Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization; **Riccardo Marchesin:** Writing – original draft, Validation, Methodology, Investigation, Formal analysis, Conceptualization; **Roberto Zunino:** Writing – review & editing, Writing – original draft, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by project SERICS (PE00000014) under the MUR National Recovery and Resilience Plan funded by the European Union – NextGenerationEU, and by PRIN 2022 PNRR project DeLiCE (F53D23009130001).

We dedicate this paper to Giangi, who inspired us to pursue theory without losing sight of practice — from which we often cheerfully stray.

Data availability

No data was used for the research described in the article.

Appendix A. Supplementary material for Section 3

In this and the following appendices we provide detailed proofs for all your statements. These proofs are presented in the order in which the statements appear in the paper, even though this order does not always reflect their logical dependencies. To clarify the relationship among our statements, Fig. A.1 displays a graph of the dependencies: an arrow $a \to b$ means that the proof of statement a depends on statement b. Note that this graph is acyclic.

Proof of Proposition 1. We have to prove that $\mathrm{MEV}_{\mathbb{A}_c}(S, \mathbb{A}_c) \geq \mathrm{MEV}(S)$. Let $\vec{\mathcal{X}} \in \kappa(\mathcal{M})^*$. Since transactions cannot create or destroy wealth *globally*, i.e. on the entire universe of accounts $\mathbb{A}$, we have that:

$$0 = \gamma_{\mathbb{A}}(S, \vec{\mathcal{X}}) = \gamma_{\mathbb{A}_c}(S, \vec{\mathcal{X}}) + \gamma_{\mathbb{A}_u \setminus \mathcal{M}}(S, \vec{\mathcal{X}}) + \gamma_{\mathcal{M}}(S, \vec{\mathcal{X}}) \tag{A.1}$$

Note that $\gamma_{\mathbb{A}_u \setminus \mathcal{M}}(S, \vec{\mathcal{X}}) \geq 0$, since a transaction crafted by $\mathcal{M}$ cannot steal tokens from user accounts. Therefore, by (A.1) we obtain $\gamma_{\mathcal{M}}(S, \vec{\mathcal{X}}) \leq -\gamma_{\mathbb{A}_c}(S, \vec{\mathcal{X}})$. Noting that $\kappa_{\mathbb{A}_c}(\mathcal{M}) = \kappa(\mathcal{M})$, we conclude:

$$\begin{aligned} \mathrm{MEV}_{\mathbb{A}_c}(S, \mathbb{A}_c) &= \max \left\{ -\gamma_{\mathbb{A}_c}(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa_{\mathbb{A}_c}(\mathcal{M})^* \right\} \\ &\geq \max \left\{ \gamma_{\mathcal{M}}(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa(\mathcal{M})^* \right\} = \mathrm{MEV}(S) \end{aligned}$$

□

Proof of Lemma 2. For item 1 we need to prove the following equalities.

a. $\mathrm{MEV}_D(S, \emptyset) = 0$. Since there are no victims, the identity $\gamma_\emptyset(S, \vec{\mathcal{X}}) = 0$ holds for any $\vec{\mathcal{X}}$.

b. $\mathrm{MEV}_\emptyset(S, C) = 0$. Since $\kappa_\emptyset(\mathcal{M})$ is empty, the only $\vec{\mathcal{X}} \in \kappa_\emptyset(\mathcal{M})^*$ is the empty sequence ε, which preserve the state. Then, $\gamma_C(S, \varepsilon) = \$_C(S) - \$_C(S) = 0$.

For item 2, we prove two inequalities:

a. Let $\varepsilon \in \kappa_D(\mathcal{M})^*$ be the empty sequence of transactions. We have that:

$$0 \;=\; \gamma_C(S, \varepsilon) \leq \max \left\{ -\gamma_C(S, \vec{\mathcal{X}}) \,\middle|\, \vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^* \right\} \;=\; \mathrm{MEV}_D(S, C)$$

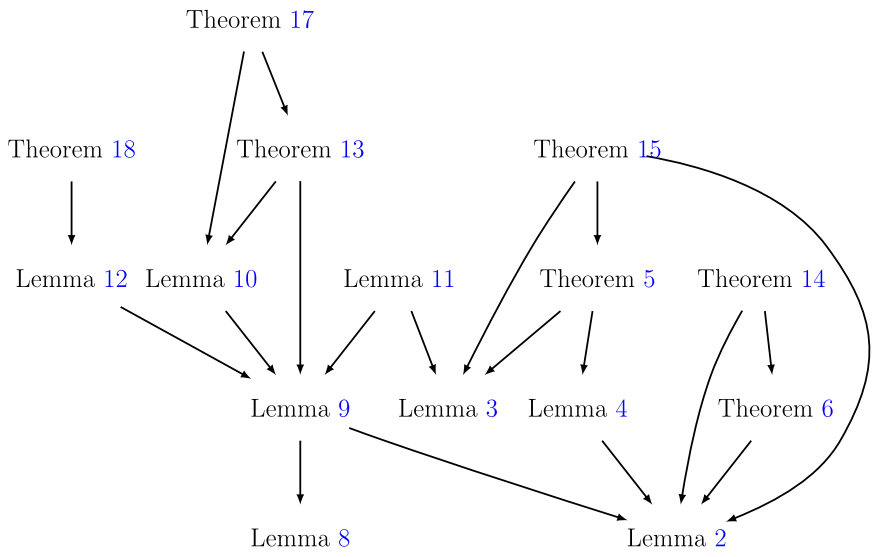


Fig. A.1. Dependencies among the statements.

b. The amount of tokens that C can lose is bounded by the amount of tokens contained in it, and so $\gamma_C(S,\vec{\mathcal{X}}) \geq -\$_C(S)$. This implies $-\gamma_C(S,\vec{\mathcal{X}}) \leq \$_C(S)$, and since this holds for all $\vec{\mathcal{X}}$ we have the thesis.

For item (3) we first prove $\mathrm{MEV}_D(S,C) = \mathrm{MEV}_D(S, C \cap \dagger S)$. For all S and $\vec{\mathcal{X}}$:

$$\gamma_C(S,\vec{\mathcal{X}}) \;=\; \sum_{\mathtt{C}\in C} \gamma_{\mathtt{C}}(S,\vec{\mathcal{X}}) \;=\; \sum_{\mathtt{C}\in C\cap\dagger\Gamma} \gamma_{\mathtt{C}}(S,\vec{\mathcal{X}}) \;=\; \gamma_{C\cap\dagger\Gamma}(S,\vec{\mathcal{X}})$$

where the second equality holds because any transaction in $\vec{\mathcal{X}}$ targeting a contract $\mathtt{D} \notin \dagger S$ is invalid, causing $\gamma_{\mathtt{D}}(S,\vec{\mathcal{X}}) = 0$. Since this holds for any $\vec{\mathcal{X}}$, the first equality of item 3 is proven. We now prove $\mathrm{MEV}_D(S,C) = \mathrm{MEV}_{D\cap\dagger S}(S,C)$. To do so, just note that a transaction in $\kappa_D(\mathcal{M})$ that calls a contract not appearing in S is invalid, and therefore will have no effect on the loss of C.

For item 4, the proof follows directly from the inclusion $\kappa_D(\mathcal{M}) \subseteq \kappa_{D'}(\mathcal{M})$.

For item (5), Let $\vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^*$ be a sequence of transactions that maximizes the loss $-\gamma_C(S,\vec{\mathcal{X}})$. Without loss of generality, assume that $\vec{\mathcal{X}}$ is valid in S: in particular, this implies that $\vec{\mathcal{X}}$ never triggers calls (either external or internal) to methods of contracts not in S (otherwise, such call would revert, and the sequence would not be valid). Therefore, the contract state Δ is not affected by the execution of $\vec{\mathcal{X}}$, that is:

$$S \xrightarrow{\vec{\mathcal{X}}} S' \implies S \mid \Delta \xrightarrow{\vec{\mathcal{X}}} S' \mid \Delta$$

Therefore:

$$\begin{aligned}\gamma_C(S \mid \Delta, \vec{\mathcal{X}}) &= \$_C(S' \mid \Delta) - \$_C(S \mid \Delta)\\ &= \$_C(S') + \$_C(\Delta) - (\$_C(S) + \$_C(\Delta))\\ &= \$_C(S') - \$_C(S)\\ &= \gamma_C(S,\vec{\mathcal{X}})\end{aligned}$$

which shows the inequality $\mathrm{MEV}_D(S,C) \leq \mathrm{MEV}_D(S \mid \Delta, C)$. □

Proof of Lemma 3. Let $D \subseteq \dagger S$. We must prove that $\mathrm{MEV}_D(S \mid \Delta, C) = \mathrm{MEV}_D(S,C)$. The inequality $\geq$ follows from item 5 of Lemma 2. For the inequality $\leq$, let $\vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^*$ be a valid sequence of transactions that maximizes $-\gamma_C(S \mid \Delta, \vec{\mathcal{X}})$. We show that $\vec{\mathcal{X}}$ causes the same loss to C in S. Since the transactions in $\vec{\mathcal{X}}$ target contracts in $D \subseteq \dagger S$ and since, by the well-formedness of S, the contracts in S have no dependencies in Δ, then the contracts in Δ are not affected by $\vec{\mathcal{X}}$. Hence, executing $\vec{\mathcal{X}}$ yields a transition of the form:

$$S \mid \Delta \xrightarrow{\vec{\mathcal{X}}} S' \mid \Delta$$

Since $\vec{\mathcal{X}}$ does not include any direct/indirect calls to the contracts in Δ, then $\vec{\mathcal{X}}$ is also valid in S, producing the same effect as in $S \mid \Delta$, namely:

$$S \xrightarrow{\vec{\mathcal{X}}} S'$$

To prove that the loss is preserved, observe that:

$$\begin{aligned}\gamma_C(S, \vec{\mathcal{X}}) &= \$_C(S') - \$_C(S)\\ &= \$_C(S') + \$_C(\Delta) - \$_C(S) - \$_C(\Delta)\\ &= \$_C(S' \mid \Delta) - \$_C(S \mid \Delta)\\ &= \gamma_C(S \mid \Delta, \vec{\mathcal{X}})\end{aligned}$$

This implies the thesis:

$$\mathrm{MEV}_D(S \mid \Delta, C) \le \mathrm{MEV}_D(S, C)$$

□

Proof of Lemma 4. First, note that the inequality $\mathrm{MEV}_{D \cap deps(C)}(S, C) \le \mathrm{MEV}_D(S, C)$ follows from item 4 of Lemma 2, so we just need to show that:

$$\mathrm{MEV}_D(S, C) \le \mathrm{MEV}_{D \cap deps(C)}(S, C)$$

Let $\vec{\mathcal{X}} \in \kappa_D(\mathcal{M})^*$ be a sequence of transactions that maximizes the loss of C when executed in state S. We show that there exists $\vec{\mathcal{Y}} \in \kappa_{D \cap deps(C)}(\mathcal{M})^*$ that causes a loss to C equal to the one caused by $\vec{\mathcal{X}}$, namely:

$$\vec{\mathcal{Y}} \in \kappa_{D \cap deps(C)}(\mathcal{M})^* \qquad \gamma_C(S, \vec{\mathcal{Y}}) = \gamma_C(S, \vec{\mathcal{X}}) \tag{A.2}$$

W.l.o.g. we assume that all the transactions in $\vec{\mathcal{X}}$ are valid: indeed, invalid transactions in $\vec{\mathcal{X}}$ are reverted, so they can be removed without affecting the loss.

Note that each transaction $\mathtt{X}_i = \mathtt{M}_\mathtt{i} : \mathtt{D}_{\mathtt{i},1}.\mathtt{f}_{\mathtt{i},1}(\mathtt{args}_{\mathtt{i},1})$ in $\vec{\mathcal{X}}$ can trigger a sequence of *internal* contract-to-contract method calls:

$$\mathtt{C}_{\mathtt{i},1} : \mathtt{D}_{\mathtt{i},1}.\mathtt{f}_{\mathtt{i},1}(\mathtt{args}_{\mathtt{i},1})\ \ \mathtt{C}_{\mathtt{i},2} : \mathtt{D}_{\mathtt{i},2}.\mathtt{f}_{\mathtt{i},2}(\mathtt{args}_{\mathtt{i},2})\ \ \cdots\ \ \mathtt{C}_{\mathtt{i},\mathtt{k}} : \mathtt{D}_{\mathtt{i},\mathtt{k}}.\mathtt{f}_{\mathtt{i},\mathtt{k}}(\mathtt{args}_{\mathtt{i},\mathtt{k}})$$

Let $\vec{x}$ be the sequence of all method calls (either external or internal) that are performed upon the execution of $\vec{\mathcal{X}}$ in state S. To construct $\vec{\mathcal{Y}}$, we start by considering the subsequence $\vec{y}$ of $\vec{x}$ containing all and only the calls of the form:

(A) $\mathtt{M}_\mathtt{i} : \mathtt{D}_{\mathtt{i},1}.\mathtt{f}_{\mathtt{i},1}(\mathtt{args}_{\mathtt{i},1})$ where $\mathtt{D}_{\mathtt{i},1} \in deps(C)$, or
(B) $\mathtt{C}_{\mathtt{i},\mathtt{j}} : \mathtt{D}_{\mathtt{i},\mathtt{j}}.\mathtt{f}_{\mathtt{i},\mathtt{j}}(\mathtt{args}_{\mathtt{i},\mathtt{j}})$, where $\mathtt{C}_{\mathtt{i},\mathtt{j}} \notin deps(C)$ and $\mathtt{D}_{\mathtt{i},\mathtt{j}} \in deps(C)$.

Claim (1). If $\mathtt{C}_{\mathtt{i},\mathtt{j}} : \mathtt{D}_{\mathtt{i},\mathtt{j}}.\mathtt{f}_{\mathtt{i},\mathtt{j}}(\mathtt{args}_{\mathtt{i},\mathtt{j}}) \in \vec{y}$, then $\mathtt{D}_{\mathtt{i},\mathtt{j}} \in \partial(C, D)$.

Proof. By hypothesis, $\mathtt{D}_{\mathtt{i},\mathtt{j}} \in deps(C)$ and $\mathtt{C}_{\mathtt{i},\mathtt{j}}$ calls $\mathtt{D}_{\mathtt{i},\mathtt{j}}$. Since the first contract called in the transaction, that is $\mathtt{D}_{\mathtt{i},1}$, belongs to D, then $\mathtt{C}_{\mathtt{i},\mathtt{j}} \in deps(D)$. By hypothesis, $\mathtt{C}_{\mathtt{i},\mathtt{j}} \notin deps(C)$. Therefore, $\mathtt{C}_{\mathtt{i},\mathtt{j}} \in deps(D) \setminus deps(C)$. By definition of $\partial(C, D)$, this concludes the proof of Claim (1).

To describe the construction of $\vec{\mathcal{Y}}$, let the meta-variables $\mathtt{a}_i$ range over user and contract addresses, so to rewrite the sequence $\vec{y}$ as follows:

$$\mathtt{a}_1 : \mathtt{C}_1.\mathtt{f}_1(\mathtt{args}_1)\ \ \mathtt{a}_2 : \mathtt{C}_2.\mathtt{f}_2(\mathtt{args}_2)\ \ \cdots\ \ \mathtt{a}_n : \mathtt{C}_\mathtt{n}.\mathtt{f}_\mathtt{n}(\mathtt{args}_\mathtt{n})\cdots$$

To construct $\vec{\mathcal{Y}}$, we translate each call $\mathtt{a}_i : \mathtt{C}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\mathtt{args}_\mathtt{i})$ in $\vec{y}$ into a transaction as follows, depending on whether the call is due to conditions (A) or (B):

A In this case, $\mathtt{a}_i = \mathtt{M}_\mathtt{i}$, hence the call $\mathtt{a}_i : \mathtt{C}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\mathtt{args}_\mathtt{i})$ is already a transaction, which we preserve in $\vec{\mathcal{Y}}$.
B In this case, we replace the contract account $\mathtt{a}_i$ in the call $\mathtt{a}_i : \mathtt{C}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\mathtt{args}_\mathtt{i})$ in $\vec{y}$ with the user account $\mathtt{M}_\mathtt{i}$ that originated the corresponding call in $\vec{y}$.

Claim (2). $\vec{\mathcal{Y}} \in \kappa_{D \cap deps(C)}(\mathcal{M})^*$

Proof. Consider a transaction $\mathtt{Y}_i$ in $\vec{\mathcal{Y}}$. We have two cases, depending on whether $\mathtt{Y}_i$ is due to conditions (A) or (B) given before:

(A) in this case, $\mathtt{Y}_i$ is equal to some $\mathtt{X}_j = \mathtt{M}_\mathtt{j} : \mathtt{D}_{\mathtt{j},1}.\mathtt{f}_{\mathtt{j},1}(\mathtt{args}_{\mathtt{j},1})$ in $\vec{\mathcal{X}}$ where $\mathtt{D}_{\mathtt{j},1} \in deps(C)$. Since $\mathtt{X}_j \in \kappa_D(\mathcal{M})$, then $\mathtt{D}_{\mathtt{j},1} \in D$, and so $\mathtt{Y}_i \in \kappa_{D \cap deps(C)}(\mathcal{M})$.
(B) by Claim (1), the callee of $\mathtt{Y}_i$ is in $\partial(C, D)$, which is included in D by assumption (2), and is included in $deps(C)$ by definition of $\partial(C, D)$. This implies that $\mathtt{Y}_i \in \kappa_{D \cap deps(C)}(\mathcal{M})$, since $\mathcal{M}$ is able to infer the actual arguments of the involved call by simulating the execution of $\vec{\mathcal{X}}$ in S. This completes the proof of Claim (2).

To conclude, we show that $\vec{\mathcal{Y}}$ is valid and modifies the state of the contracts in C in exactly the same way as $\vec{\mathcal{X}}$. Together with Claim (2), this will prove (A.2).

Observe that the sequence of calls due to the execution of $\vec{\mathcal{Y}}$ coincides with the subsequence of $\vec{x}$ including all and only the calls to contracts in $deps(C)$ — with the only difference of the change of sender due to condition (B). We show, however, that this difference does not affect neither the validity of the transactions nor their effect on the state update. Indeed, we have that:

1. Each transaction $\mathtt{Y}_i$ in $\vec{\mathcal{Y}}$ can be funded by the adversary. To show this, let $\mathtt{Y}_i$ be derived from $\mathtt{a}_i : \mathtt{C}_\mathtt{i}.\mathtt{f}_\mathtt{i}(\mathtt{args}_\mathtt{i})$. We have the following cases:
   - $\mathtt{a}_i = \mathtt{M}_\mathtt{i}$. By contradiction, assume that in the state where that transaction is executed, $\mathtt{M}_\mathtt{i}$ does not have the tokens needed to fund the call to $\mathtt{C}_\mathtt{i}$. Since the corresponding transaction $\mathtt{X}_j$ in $\vec{\mathcal{X}}$ was valid, this means that $\mathtt{M}_\mathtt{i}$ had received some of the tokens needed to fund the call from a previous transaction $\mathtt{X}_h = \mathtt{M}_\mathtt{h} : \mathtt{D}_\mathtt{h}.\mathtt{f}_\mathtt{h}(\mathtt{args}_\mathtt{h})$ in $\vec{\mathcal{X}}$ that has no counterpart in $\vec{\mathcal{Y}}$. Note that:
     - $\mathtt{D}_\mathtt{h} \in D \setminus deps(C)$ because condition (A) is false for $\mathtt{X}_h$;
     - $\mathtt{C}_\mathtt{i} \in D \cap deps(C)$ by Claim (2).

     By assumption (3), there is no token flow from $D \setminus deps(C)$ to $D \cap deps(C)$ in S, which implies that no token that $\mathtt{X}_h$ has sent to $\mathtt{M}_\mathtt{i}$ is needed to call $\mathtt{C}_\mathtt{i}$ — contradiction.
   - $\mathtt{a}_i = \mathtt{D}_\mathtt{i}$. We have that:
     - $\mathtt{D}_\mathtt{i} \in deps(D) \setminus deps(C)$. The fact that $\mathtt{D}_\mathtt{i} \in deps(D)$ holds because it derives from a transaction in $\kappa_D(\mathcal{M})$, and $\mathtt{D}_\mathtt{i} \notin deps(C)$ follows by condition (B);
     - $\mathtt{C}_\mathtt{i} \in \partial(C, D)$ by Claim (1).

     By assumption (4), there is no direct token flow from $deps(D) \setminus deps(C)$ to $\partial(C, D)$ in S. Therefore, there is no token transfer from $\mathtt{D}_\mathtt{i}$ to $\mathtt{C}_\mathtt{i}$, and so $\mathtt{Y}_i$ does not need to be funded.
2. The transactions $\mathtt{Y}_i$ that are in $\vec{\mathcal{Y}}$ due to condition (B) have callee in $\partial(C, D)$ by Claim (1), and so their methods are *sender-agnostic* by assumption (1). So, the fact that in the execution of $\mathtt{Y}_i$ they are called directly from a user address, while in the execution of $\mathtt{X}_i$ they are called from a contract address, does not affect the execution of these calls. Note that the adversary could receive tokens from these calls, namely those tokens which are transferred to the sender, but from the previous item such tokens are irrelevant to the validity of $\vec{\mathcal{Y}}$.
3. The transaction $\mathtt{M}_\mathtt{i} : \mathtt{C}_{\mathtt{i},\mathtt{j}}.\mathtt{f}_{\mathtt{i},\mathtt{j}}(\mathtt{args}_{\mathtt{i},\mathtt{j}})$ that are in $\vec{\mathcal{Y}}$ due to condition (B) may transfer tokens to $\mathtt{M}_\mathtt{i}$, while the corresponding calls $\mathtt{C}_{\mathtt{i},\mathtt{j}-1} : \mathtt{C}_{\mathtt{i},\mathtt{j}}.\mathtt{f}_{\mathtt{i},\mathtt{j}}(\mathtt{args}_{\mathtt{i},\mathtt{j}})$ due to the execution of $\vec{\mathcal{X}}$ may send these tokens to the sender $\mathtt{C}_{\mathtt{i},\mathtt{j}-1}$, thus affecting its gain. This difference however does not affect the loss of C, since $\mathtt{C}_{\mathtt{i},\mathtt{j}-1}$ is not in $deps(C)$ by condition (B).

For the reasons above, $\vec{\mathcal{Y}}$ is valid, and it updated the state of contracts $deps(C)$ in exactly the same way as $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$ — and, in particular, $\vec{\mathcal{X}}$ and $\vec{\mathcal{Y}}$ cause exactly the same losses to the contracts in C, i.e. $\gamma_C(S, \vec{\mathcal{Y}}) = \gamma_C(S, \vec{\mathcal{X}})$. This proves (A.2), from which we obtain the thesis. □

Example 20. To illustrate the need of the sender-agnosticism assumption (1) in Lemma 4, consider the contracts:

```
contract C0 { function f() { require (msg.sender==C1); T.send(M,1); }}
contract C1 { function f() { C0.f(); } }
```

Let $\mathcal{M} = \{\mathtt{M}\}$, $D = \{\mathtt{C0}, \mathtt{C1}\}$, $C = \{\mathtt{C0}\}$, and let:

$$S = \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{C0}[1 : \mathtt{T}] \mid \mathtt{C1}[0 : \mathtt{T}]$$

Let $\vec{\mathcal{X}} = \mathtt{M} : \mathtt{C1.f}() \in \kappa_D(\mathtt{M})$. By executing $\vec{\mathcal{X}}$ in S, we have that:

$$S \xrightarrow{\mathtt{M:C1.f()}} \mathtt{M}[1 : \mathtt{T}] \mid \mathtt{C0}[0 : \mathtt{T}] \mid \mathtt{C1}[0 : \mathtt{T}]$$

Since there are no tokens left in C, $\vec{\mathcal{X}}$ clearly maximises the loss of C, hence $\mathrm{MEV}_D(S, C) = 1 \cdot \$\mathbf{1}_\mathtt{T}$. Since $\mathtt{C0} \in \partial(C, D)$ is not sender-agnostic, narrowing the set of callable contracts to $D \cap deps(C) = \{\mathtt{C0}\}$ is not guaranteed to preserve the MEV. Indeed, $\mathrm{MEV}_{\{\mathtt{C0}\}}(S, C) = 0$, since the adversary can only call $\mathtt{C0}$, but the transaction would revert since the require condition in $\mathtt{C0}$ is violated. ◇

Example 21. To illustrate the need of the assumption (2) in Lemma 4, consider the contracts:

```
contract C0 { function f() { T.send(M,1); } }
contract C1 { function f() { C0.f(); } }
```

Let $\mathcal{M} = \{\mathtt{M}\}$, $D = \{\mathtt{C1}\}$, $C = \{\mathtt{C0}\}$, and let:

$$S = \mathtt{M}[0 : \mathtt{T}] \mid \mathtt{C0}[1 : \mathtt{T}] \mid \mathtt{C1}[0 : \mathtt{T}]$$

Let $\vec{\mathcal{X}} = \mathtt{M} : \mathtt{C1.f}() \in \kappa_D(\mathtt{M})$. By executing $\vec{\mathcal{X}}$ in S, we have that:

$$S \xrightarrow{\mathtt{M:C1.f()}} \mathtt{M}[1 : \mathtt{T}] \mid \mathtt{C0}[0 : \mathtt{T}] \mid \mathtt{C1}[0 : \mathtt{T}]$$

Since there are no tokens left in C, $\vec{\mathcal{X}}$ clearly maximises the loss of C, hence $\mathrm{MEV}_D(S, C) = 1 \cdot \$\mathbf{1}_\mathtt{T}$. Note that conditions (1), (3) and (4) are trivially satisfied, while $\partial(C, D) = \{\mathtt{C0}\} \not\subseteq D$, thus violating condition (2). Therefore, narrowing the set of callable contracts to $D \cap deps(C) = \emptyset$ does not preserve the MEV: indeed, such MEV is zero, since there are no callable contracts. ◇

Example 22. To illustrate the need of the token flows assumptions in Lemma 4, consider the following contracts:

```
contract C0 { function f(){ T.receive(2); T.send(M,4); } }
contract C1 { function f(){ T.receive(1); T.send(C0,2); C0.f(); } }
contract C2 { function f(){ T.send(C1,1); C1.f(); } }
```

Let $\mathcal{M} = \{\mathtt{M}\}$, $D = \{\mathtt{C0}, \mathtt{C1}, \mathtt{C2}\}$, $C = \{\mathtt{C0}, \mathtt{C1}\}$, and let:

$$S \;=\; \mathtt{M}[0:\mathtt{T}] \mid \mathtt{C0}[2:\mathtt{T}] \mid \mathtt{C1}[1:\mathtt{T}] \mid \mathtt{C2}[1:\mathtt{T}]$$

Let $\vec{\mathscr{X}} = \mathtt{M} : \mathtt{C2.f}() \in \kappa_D(\mathtt{M})$. By executing $\vec{\mathscr{X}}$ in S, we obtain:

$$S \xrightarrow{\mathtt{M}:\mathtt{C3.f}()} \mathtt{M}[4:\mathtt{T}] \mid \mathtt{C0}[0:\mathtt{T}] \mid \mathtt{C1}[0:\mathtt{T}] \mid \mathtt{C2}[0:\mathtt{T}]$$

Since there are no tokens left in C, $\vec{\mathscr{X}}$ clearly maximises the loss of C. Hence, $\mathrm{MEV}_D(S, C) = 3 \cdot \$\mathbf{1}_\mathtt{T}$. Similarly to Example 8, we have that $\partial(C, D) = \{\mathtt{C1}\}$, which satisfies conditions (1) and (2) of Lemma 4. Instead, condition (4) is violated, since there is a direct token flow of $1 : \mathtt{T}$ from $\mathtt{C2} \in deps(D) \setminus deps(C)$ to $\mathtt{C1} \in \partial(C, D)$. Therefore, narrowing the set of callable contracts to $D \cap deps(C) = \{\mathtt{C0}, \mathtt{C1}\}$ is not guaranteed to preserve the MEV. Indeed, $\mathtt{M}$ has no tokens in S: hence, calling $\mathtt{C0}$ fails, since $\mathtt{C0}$ has not the required $4 : \mathtt{T}$ to transfer; similarly, calling $\mathtt{C1}$ fails, since $\mathtt{C1}$ does not have the $2 : \mathtt{T}$ required to call $\mathtt{C0}$. Therefore, $\mathrm{MEV}_{\{\mathtt{C0},\mathtt{C1}\}}(S, C) = 0 < \mathrm{MEV}_D(S, C)$. ◇

Appendix B. Supplementary material for Section 4

Proof of Lemma lem:mev:wallet. For Item 1, let $\mathcal{M} \cap \mathrm{dom}\, W = \emptyset$. Since

- $\mathcal{M}$ cannot spend tokens from the wallets in W, and
- the presence or absence of a user account is irrelevant to the effect transactions originated from other accounts

, then any sequence of transactions $\vec{\mathscr{X}} \in \kappa_D(\mathcal{M})^*$ is valid in S if and only if it is valid in $S \mid W$. By wallet monotonicity (Lemma 7), if $\vec{\mathscr{X}}$ is valid (in either state), then

$$S \xrightarrow{\vec{\mathscr{X}}} W' \mid \Gamma' \implies S \mid W \xrightarrow{\vec{\mathscr{X}}} (W' + W) \mid \Gamma'$$

From this, we obtain that $\vec{\mathscr{X}}$ causes the same loss to C in both S and $S \mid W$:

$$\begin{aligned}\gamma_C(S, \vec{\mathscr{X}}) &= \$_C(W' \mid \Gamma') - \$_C(S) \;=\; \$_C(\Gamma') - \$_C(S) \\ &= \$_C(W' + W \mid \Gamma') - \$_C(S \mid W) \;=\; \gamma_C(S \mid W, \vec{\mathscr{X}})\end{aligned}$$

Therefore, we have the thesis $\mathrm{MEV}_D(S, C) = \mathrm{MEV}_D(S \mid W, C)$.

For item 2, let $\vec{\mathscr{X}} \in \kappa_D(\mathcal{M})^*$ maximize the loss of C. Without loss of generality, assume that $\vec{\mathscr{X}}$ is valid in S. By wallet monotonicity, $\vec{\mathscr{X}}$ is also valid in S' and it extracts the same amount of tokens from C, giving us a lower bound for $\mathrm{MEV}_D(S', C)$. □

Proof of Lemma lem:mev:stability. Fix C, D and Γ, and let $f(w) = \mathrm{MEV}_D(w \mid \Gamma, C)$. By item 2 of Lemma 2, $f(w)$ is bounded from above by $\$_C(w \mid \Gamma)$. By Definition 1, the wealth $\$_C(w \mid \Gamma)$ only depends on the contract state Γ, so it is equal to $\$_C(\Gamma)$. Hence, $f(w)$ is an integer value bounded from above, so there exists a wallet W for which the maximum is reached. By item 2 of Lemma 8, the same MEV is also achieved with a wallet extension W' of W. □

Proof of Lemma lem:rich-mev:basic. Items 1 and 3 to 5 have an analogous statement in Lemma 2, which holds for any wallet state. Due to the stability property (Lemma 9) and the definition of MEV^∞ (Definition 7), the wealthy-adversary versions of the statements must also hold. The same reasoning can be carried out for Item 2 and its analogous in Lemma 2, since $\$_C(\Gamma) = \$_C(W \mid \Gamma)$ for any W. □

Appendix C. Supplementary material for Section 5

Proof of Proposition lem:rich-nonint-nonint. From Definition 7 and Lemma 9, there exist wallets W_1 and W_2 such that:

$$\forall W \geq_\$ W_1 \;:\; \mathrm{MEV}^\infty_{\dagger\Delta}(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}_{\dagger\Delta}(W \mid \Gamma \mid \Delta, \dagger\Delta) \tag{C.1}$$

$$\forall W \geq_\$ W_2 \;:\; \mathrm{MEV}^\infty(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}(W \mid \Gamma \mid \Delta, \dagger\Delta) \tag{C.2}$$

Let $W_0 = \sup\{W_1, W_2\}$. Then, for all $W \geq_\$ W_0$:

$$\begin{aligned}&\Gamma \not\rightsquigarrow^\infty \Delta \\ \iff\; &\mathrm{MEV}^\infty_{\dagger\Delta}(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}^\infty(\Gamma \mid \Delta, \dagger\Delta) && \text{by def. } \not\rightsquigarrow^\infty\end{aligned}$$

$\iff \mathrm{MEV}_{\dagger\Delta}(W \mid \Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}(W \mid \Gamma \mid \Delta, \dagger\Delta)$ by (C.1) and (C.2)

$\iff W \mid \Gamma \not\rightsquigarrow \Delta$ by def. $\not\rightsquigarrow$

□

Appendix D. Supplementary material for Section 10

Example 23. $\Gamma \not\rightsquigarrow^\infty \Delta$ does not imply $\Gamma \not\rightsquigarrow^\infty \Delta \mid \Delta'$. This can be simply seen when $\Delta = \emptyset$ (which is trivially non-interfering with any Γ), and Δ' is any contract that interferes with Γ. For instance, let

$$\Gamma = \mathtt{X}[0 : \mathtt{T}, \mathtt{x} = 0] \quad \Delta = \emptyset \quad \Delta' = \mathtt{C}[1 : \mathtt{T}]$$

where $\mathtt{X}$ and $\mathtt{C}$ are defined in Fig. 19. We have that

$$\mathrm{MEV}^\infty_{\dagger\Delta}(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}^\infty_{\emptyset}(\Gamma \mid \Delta, \emptyset) = 0$$

$$\mathrm{MEV}^\infty(\Gamma \mid \Delta, \dagger\Delta) = \mathrm{MEV}^\infty(\Gamma \mid \Delta, \emptyset) = 0$$

so $\Gamma \not\rightsquigarrow^\infty \Delta$. To study the non-interference between Γ and $\Delta \mid \Delta'$, we note that the *restricted* local MEV of $\mathtt{C}$ is 0: indeed, if the adversary cannot call methods of $\mathtt{X}$, then they have no way to change the value of the variable $\mathtt{x}$, and the method $\mathtt{C.f()}$ will never be enabled. On the other hand, if the adversary has access to $\mathtt{X}$, she can perform the following sequence of transactions: $\mathtt{M} : \mathtt{X.set}(1)\ \mathtt{M} : \mathtt{C.f}()$, extracting $1 : \mathtt{T}$ from $\mathtt{C}$. Therefore, $\Gamma \rightsquigarrow^\infty \Delta \mid \Delta'$. Note that since the methods of Δ' do not depend on Δ, we can change their order while preserving the well formedness of the state, so $\Gamma \rightsquigarrow^\infty \Delta' \mid \Delta$.

Example 24. $\Gamma \not\rightsquigarrow^\infty \Delta \mid \Delta'$ does not imply $\Gamma \not\rightsquigarrow^\infty \Delta$. For instance, let

$$\Gamma = \mathtt{X}[0 : \mathtt{T}, \mathtt{x} = 0] \quad \Delta = \mathtt{C}[1 : \mathtt{T}] \quad \Delta' = \mathtt{ForwardX}[0 : \mathtt{T}]$$

using the contracts in Listing 19. From Example 23 we know that $\Gamma \rightsquigarrow^\infty \Delta$. However, $\Gamma \not\rightsquigarrow^\infty \Delta \mid \Delta'$. Indeed, it does not matter if the adversary has direct access to $\mathtt{X}$, since it can always call $\mathtt{ForwardX}$ to maximize the loss of $\Delta \mid \Delta'$. More specifically, the sequence of transactions $\mathtt{M} : \mathtt{ForwardX.set_x}(1)\ \mathtt{M} : \mathtt{C.f}()$ causes a loss of $1 : \mathtt{T}$ in $\mathtt{C}$, so

$$\mathrm{MEV}^\infty_{\dagger(\Delta \mid \Delta')}(\Gamma \mid \Delta \mid \Delta', \dagger(\Delta \mid \Delta')) = 1.$$

The unrestricted MEV is also equal to 1 (it cannot be greater, since $\$(\Delta \mid \Delta') = 1$), so we have

$$\mathrm{MEV}^\infty_{\dagger(\Delta \mid \Delta')}(\Gamma \mid \Delta \mid \Delta', \dagger(\Delta \mid \Delta')) = 1 = \mathrm{MEV}^\infty(\Gamma \mid \Delta \mid \Delta', \dagger(\Delta \mid \Delta'))$$

and $\Gamma \not\rightsquigarrow^\infty \Delta \mid \Delta'$. Since there are no calls from Δ' to Δ, we can also change their order while preserving the well formedness of the state, and conclude that $\Gamma \not\rightsquigarrow^\infty \Delta' \mid \Delta$.

```
contract X {
  uint public x;
  constructor { x=0; }
  function set(uint y) public { x=y; }
}

contract C {
  function f() { require (X.x()==1); T.send(msg.sender,1); }
}

contract ForwardX {
  function get_x()  { return X.x(); }
  function set_x(y) { X.set(y); }
}
```

Listing 19. Contracts for Examples 23 and 24.

```
contract Var {
  uint public x;
  function set(uint y) public { if (x==0) x=y; }
}

contract Drop {
  uint b=0;
  function drop2() { require (b==0 && Var.x()==1); b=1; T.send(msg.sender,2); }
  function drop3() { require (b==0 && Var.x()==0); b=1; Var.set(2); T.send(msg.sender,3); }
}
```

Listing 20. Contracts for Example 25.

Example 25. $\Gamma \not\rightsquigarrow^{\infty} \Delta_1$ and $\Gamma \not\rightsquigarrow^{\infty} \Delta_2$ does not imply $\Gamma \not\rightsquigarrow^{\infty} \Delta_1 \mid \Delta_2$. Let:

$$\Gamma = \texttt{Var}[0 : \texttt{T}, \texttt{x} = 0] \quad \Delta_1 = \texttt{Drop1}[3 : \texttt{T}, \texttt{b} = 0] \quad \Delta_2 = \texttt{Drop2}[3 : \texttt{T}, \texttt{b} = 0]$$

using the contracts in Listing 20, where `Drop` is deployed twice in Γ. The wealthy adversaries' MEV of both instances `Drop1` and `Drop2` in Γ is 3, obtained by calling `drop3` exactly once. This holds both for the unrestricted and restricted MEV, hence $\Gamma \not\rightsquigarrow^{\infty} \Delta_1$ and $\Gamma \not\rightsquigarrow^{\infty} \Delta_2$.

To study the composition of Γ with $\Delta_1 \mid \Delta_2$, note that the unrestricted MEV of $\{\texttt{Drop1}, \texttt{Drop2}\}$ is 4, which can be obtained by the sequence:

```
M : Var.set(1) M : Drop1.drop2() M : Drop2.drop2()
```

Instead the *restricted* MEV of $\{\texttt{Drop1}, \texttt{Drop2}\}$ is 3, since calling `Var` is forbidden, and `drop3` can only be called on one of the instances. Then, $\Gamma \not\rightsquigarrow^{\infty} \Delta_1 \mid \Delta_2$. ◇

References

[1] DeFi pulse: what is DeFi? Understanding decentralized finance, 2019 . www.defipulse.com/blog/what-is-defi.
[2] S. Werner, D. Perez, L. Gudgeon, A. Klages-Mundt, D. Harz, W.J. Knottenbelt, SoK: decentralized finance (DeFi), in: ACM Conference on Advances in Financial Technologies, (AFT), ACM, 2022, pp. 30–46. https://doi.org/10.1145/3558535.3559780
[3] S. Kitzler, F. Victor, P. Saggese, B. Haslhofer, A systematic investigation of DeFi compositions in ethereum, in: Financial Cryptography and Data Security Workshops, Vol. 13412 of *LNCS*, Springer, 2022, pp. 272–279. https://doi.org/10.1007/978-3-031-32415-4_18
[4] S. Kitzler, F. Victor, P. Saggese, B. Haslhofer, Disentangling decentralized finance (DeFi) compositions, ACM Trans. Web 17 (2) (2023) 10:1–10:26. https://doi.org/10.1145/3532857
[5] 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: IEEE Symp. on Security and Privacy, IEEE, 2020, pp. 910–927. https://doi.org/10.1109/SP40000.2020.00040
[6] L. Gudgeon, D. Pérez, D. Harz, B. Livshits, A. Gervais, The decentralized financial crisis, in: Crypto Valley Conference on Blockchain Technology (CVCBT), IEEE, 2020, pp. 1–15. https://doi.org/10.1109/CVCBT50464.2020.00005
[7] FuruCombo website, 2023, https://furucombo.app.
[8] K. Babel, P. Daian, M. Kelkar, A. Juels, Clockwork finance: automated analysis of economic security in smart contracts, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, 2023, pp. 622–639. https://doi.org/10.1109/SP46215.2023.00036
[9] J.A. Goguen, J. Meseguer, Security policies and security models, in: IEEE Symposium on Security and Privacy, IEEE Computer Society, 1982, pp. 11–20. https://doi.org/10.1109/SP.1982.10014
[10] P.Y.A. Ryan, S.A. Schneider, Process algebra and non-Interference, in: IEEE Computer Security Foundations Workshop, IEEE Computer Society, 1999, pp. 214–227. https://doi.org/10.1109/CSFW.1999.779775
[11] P.Y.A. Ryan, J.D. McLean, J.K. Millen, V.D. Gligor, Non-interference: who needs it?, in: IEEE Computer Security Foundations Workshop, IEEE Computer Society, 2001, pp. 237–238. https://doi.org/10.1109/CSFW.2001.930149
[12] M. Backes, B. Pfitzmann, Computational probabilistic non-interference, in: European Symposium on Research in Computer Security (ESORICS), Vol. 2502 of *LNCS*, Springer, 2002, pp. 1–23. https://doi.org/10.1007/3-540-45853-0_1
[13] R. Giacobazzi, I. Mastroeni, Abstract non-interference: parameterizing non-interference by abstract interpretation, in: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, 2004, pp. 186–197. https://doi.org/10.1145/964001.964017
[14] M. Bartoletti, R. Zunino, A theoretical basis for MEV, in: Financial Cryptography and Data Security, LNCS, Springer, 2025, pp. 225–242. https://doi.org/10.1007/978-3-032-07035-7_14
[15] K. Qin, L. Zhou, B. Livshits, A. Gervais, Attacking the DeFi ecosystem with flash loans for fun and profit, in: Financial Cryptography, Vol. 12674 of *LNCS*, Springer, 2021, pp. 3–32. https://doi.org/10.1007/978-3-662-64322-8_1
[16] Flashbots, 2021, flashbots.net.
[17] S. Blackshear, E. Cheng, D.L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, D. Rain, S. Russi, T. Sezer, R. Zakian, Zhou, Move: a language with programmable resources, 2019. https://api.semanticscholar.org/CorpusID:201681125.
[18] G. Angeris, T. Chitra, Improved price oracles: constant function market makers, in: ACM Conference on Advances in Financial Technologies (AFT), ACM, 2020, pp. 80–91. https://arxiv.org/abs/2003.10001. https://doi.org/10.1145/3419614.3423251
[19] G. Angeris, H.-T. Kao, R. Chiang, C. Noyes, T. Chitra, An analysis of uniswap markets, Cryptoecon. Syst. 1 (1) (2021). https://doi.org/10.21428/58320208.c9738e64
[20] J. Xu, N. Vavryk, K. Paruch, S. Cousaert, SoK: decentralized exchanges (DEX) with automated market maker (AMM) protocols, ACM Comput. Surv. (2022). https://doi.org/10.1145/3570639
[21] M. Bartoletti, J.H. Chiang, A. Lluch-Lafuente, A theory of automated market makers in DeFi, Log. Methods Comput. Sci. 18 (4) (2022). https://doi.org/10.46298/lmcs-18(4:12)2022
[22] Aave v1 flashloan receiver interface, 2020, https://github.com/aave/aave-protocol/blob/efaeed363da70c64b5272bd4b8f468063ca5c361/contracts/flashloan/interfaces/IFlashLoanReceiver.sol#L11.
[23] S. Guesmi, C. Piazza, S. Rossi, Noninterference analysis for smart contracts: would you bet on it?, in: Distributed Ledger Technology Workshop (DLT), Vol. 3791 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024. https://ceur-ws.org/Vol-3791/paper12.pdf.
[24] E. Priyadarshini, M. Bartoletti, A quantitative notion of economic security for smart contract compositions, in: Financial Cryptography Workshops, Springer-Verlag, 2025, pp. 147–163. https://doi.org/10.1007/978-3-032-00492-5_11
[25] M. Bartoletti, R. Marchesin, R. Zunino, DeFi composability as MEV non-interference, CoRR (2023). https://doi.org/10.48550/ARXIV.2309.10781
[26] Understanding the DAO attack, 2025. http://www.coindesk.com/understanding-dao-hack-journalists/.
[27] A postmortem on the parity multi-sig library self-destruct, 2017 . https://goo.gl/Kw3gXi.
[28] U. Kirstein, Formal Verification Helps Finding Insolvency Bugs - Balancer V2 Bug Report, 2025. (https://medium.com/certora/formal-verification-helps-finding-insolvency-bugs-balancer-v2-bug-report-1f53ee7dd4d0). Accessed on August 30, 2024.
[29] J. Schiffl, B. Beckert, A practical notion of liveness in smart contract applications, in: Formal Methods in Blockchain (FMBC), Vol. 118 of *OASIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, pp. 8:1–8:13. https://doi.org/10.4230/OASICS.FMBC.2024.8
[30] M. Bartoletti, S. Lande, M. Murgia, R. Zunino, Verifying liquidity of recursive Bitcoin contracts, Log. Methods Comput. Sci. 18 (1) (2022). https://doi.org/10.46298/lmcs-18(1:22)2022
[31] C. Laneve, Liquidity analysis in resource-aware programming, J. Log. Algeb. Methods Program. 135 (2023) 100889. https://doi.org/10.1016/J.JLAMP.2023.100889
[32] M. Bartoletti, A. Ferrando, E. Lipparini, V. Malvone, Solvent: liquidity verification of smart contracts, colorin: Integrated Formal Methods (iFM), Vol. 15234 of *LNCS*, Springer, 2024, pp. 256–266. https://doi.org/10.1007/978-3-031-76554-4_14
[33] M. Bartoletti, L. Benetollo, M. Bugliesi, S. Crafa, G.D. Sasso, R. Pettinau, A. Pinna, M. Piras, S. Rossi, S. Salis, A. Spanò, V. Tkachenko, R. Tonelli, R. Zunino, Smart contract languages: a comparative analysis, Future Gener. Comput. Syst. 164 (2025). https://doi.org/10.1016/J.FUTURE.2024.107563

[34] N. Atzei, M. Bartoletti, S. Lande, R. Zunino, A formal model of Bitcoin transactions, in: Financial Cryptography and Data Security, Vol. 10957 of *LNCS*, Springer, 2018. https://doi.org/10.1007/978-3-662-58387-6
[35] M.M.T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M.P. Jones, P. Wadler, The extended UTXO model, in: Workshop on Trusted Smart Contracts, 2020.
[36] P. Vinogradova, O. Melkonian, Message-passing in the extended UTxO Ledger, in: Financial Cryptography Workshops, Vol. 14746 of *LNCS*, Springer, 2024, pp. 150–169. https://doi.org/10.1007/978-3-031-69231-4_11
[37] M. Bartoletti, R. Marchesin, R. Zunino, Scalable UTXO smart contracts via fine-grained distributed state, Future Gener. Comput. Syst. 175 (2026) 108023. https://doi.org/10.1016/J.FUTURE.2025.108023
[38] A. Salles, On the formalization of MEV, 2021 . https://writings.flashbots.net/research/formalization-mev.
[39] L. Zhou, K. Qin, C.F. Torres, D.V. Le, A. Gervais, High-frequency trading on decentralized on-chain exchanges, in: IEEE Symp. on Security and Privacy, IEEE, 2021, pp. 428–445. https://doi.org/10.1109/SP40001.2021.00027