TU WIEN Informatics

Simulation und Analyse von Transaktionsreihenfolgen in Ethereum

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Othmar Lechner
Matrikelnummer 11841833

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 29.08.2024

Othmar Lechner Gernot Salzer

TU WIEN Informatics

Using state changes to detect and simulate transaction order dependency in Ethereum

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Othmar Lechner
Registration Number 11841833

to the Faculty of Informatics

at the TU Wien

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

Vienna, 29.08.2024

Othmar Lechner Gernot Salzer

Declaration of Authorship

Othmar Lechner

I hereby declare that I have written this thesis independently, that I have completely specified the utilized sources and resources and that I have definitely marked all parts of the work - including tables, maps and figures - which belong to other works or to the internet, literally or extracted, by referencing the source as borrowed.

I further declare that I have used generative AI tools only as an aid, and that my own intellectual and creative efforts predominate in this work. In the appendix “Overview of Generative AI Tools Used” I have listed all generative AI tools that were used in the creation of this work, and indicated where in the work they were used. If whole passages of text were used without substantial changes, I have indicated the input (prompts) I formulated and the IT application used with its product name and version number/date.

Wien, 29.08.2024

Othmar Lechner

Acknowledgements

I want to express gratitude towards all the companions I had so far. Fooling around with siblings, hiking towards wiggly stones, sharing meals in the kitchen with flatmates, enjoying movies together, playing games, learning new languages by doing uni projects or being at random parties. This work would, and should, not exist without these precious moments.
Merci! Mulțumesc! Grazie! Gracias! Teşekkürler! جهانی سپاس! Danke! Thank you!

I also want to thank Gernot Salzer and Monika di Angelo for their continuous support. With the regular meetings and their inputs and the plentitude of constructive feedback, they helped me a lot to steadily progress and improve this work.

Kurzfassung

Ethereum speichert einen Zustand ab, der mittels Transaktionen verändert wird. Die Reihenfolge, in der Transaktionen ausgeführt werden, kann einen Einfluss auf diese Zustandsänderungen haben (transaction order dependency; TOD). Man kann analysieren, ob zwei Transaktionen TOD sind, indem man diese in zwei verschiedenen Reihenfolgen ausführt und die Ergebniszustände miteinander vergleicht. Jedoch können Transaktionen, die zwischen den beiden analysierten Transaktionen ausgeführt wurden, diese Analyse beeinflussen. Dies kann eine fokussierte Analyse von zwei Transaktionen auf TOD verhindern.

In dieser Diplomarbeit entwerfen wir eine Methode, um verschiedene Reihenfolgen von Transaktionen zu simulieren und auf TOD zu analysieren. Wir verwenden Zustandsänderungen von Transaktionen, um Zustände zu berechnen, mit denen wir verschiedene Reihenfolgen von Transaktionen simulieren. Diese Berechnungen können Zustandsänderungen von dazwischenliegenden Transaktionen inkludieren, ohne diese während der Simulation ausführen zu müssen. Weiters ermöglicht es, nur jene Änderungen am Zustand vorzunehmen, die von den Transaktionen verursacht wurden, die wir analysieren. Wir verwenden diese Simulation zum Feststellen, ob Transaktionen TOD sind und ob sie Eigenschaften eines Angriffes haben. Außerdem erläutern wir Umstände, in denen es trotz dieser Methodik zu Beeinflussungen der Analyse durch dazwischenliegende Transaktionen kommen kann.

Weiters durchsuchen wir Transaktionen, die in Ethereum ausgeführt wurden, nach Transaktionspaaren, welche potentiell TOD sind. Wir paaren Transaktionen anhand ihrer Zustandsabfragen und -änderungen. Unsere Analyse zeigt, dass nur Änderungen von Kontoständen und Kontospeichern relevant für Angriffe sind, daher verwerfen wir Paare ohne solche Änderungen. Weiters filtern wir Paare, bei welchen potentielle Störfaktoren die Simulation beeinflussen. Schließlich reduzieren wir die Anzahl an Paaren, welche ähnliche Zustandsabfragen und -änderungen haben.

Wir evaluieren unsere Methoden anhand eines existierenden Datensatzes, welcher 5.601 Angriffe aus einer Analyse von 175.552 Transaktionen enthält. Unsere Suche nach potentieller TOD markiert 5.600 Transaktionspaare als potentiell TOD. Nachdem wir diese filtern, verbleiben wir mit 115 Paaren. Wir zeigen, dass diese 115 Paare ähnlich zu 703 der gefilterten Angriffe sind. Weiters evaluieren wir unsere Simulationsmethode an allen 5.601 Angriffen und stellen bei 86% davon TOD fest und bei 81% Angriffseigenschaften. Wir analysieren die Unterschiede zwischen unseren Ergebnissen und dem Angriffsdatensatz, und führen für 60 Angriffe eine manuelle Untersuchung durch.

Abstract

In Ethereum, the order in which two transactions are executed can influence the changes they perform on the world state. One method to analyze such transaction order dependencies (TOD) is to execute the transactions in two orders and compare their behaviors. However, when simulating a reordering of two transactions, the transactions that occurred between the two transactions can influence the analysis. This influence can prevent an isolated analysis of two transactions.

To address this issue, this thesis proposes a new method to simulate transaction orders to analyze TOD. We use state changes of transactions to compute world states that we use to simulate transaction execution in different orders. This computation removes the need to execute intermediary transactions for the simulation and allows applying only the state changes of the transactions we want to analyze. We then use our simulation method to detect if transactions are TOD and show attack characteristics. We discuss cases where, despite using this method, intermediary transactions can interfere with TOD analysis.

Furthermore, we use state changes to detect transaction pairs on the blockchain that are potentially TOD. We match transactions based on the state they access and modify. By enumerating and analyzing the causes of TOD, we show that only TODs related to the storage and balance of accounts are relevant attack vectors. With this insight, we can remove matches that are irrelevant to an attack analysis. Additionally, we filter out transaction pairs where intermediary transactions may interfere with the TOD simulation. Finally, we also reduce the amount of transaction pairs with similar state accesses and modifications.

For the evaluation, we use a dataset from a previous study as a ground truth, which contains 5,601 attacks obtained from analyzing 175,552 transactions. Our method to detect potential TODs finds 5,600 of the attacks. After applying the filters, only 115 of the attacks remain for further analysis. We show that these are similar to at least 703 of the removed attacks. We apply our simulation method to all 5,601 attacks and verify that 86% of them are TOD and 81% fulfill the attack characteristic used by the ground truth. We analyze the cases where our results differ from the ground truth, including a manual analysis of 60 attacks.

Contents

1 Introduction

Ethereum is a blockchain that keeps track of a world state and updates this state by executing transactions. The transactions can execute so-called smart contracts, which are programs that are stored on the blockchain. As these programs are nearly Turing-complete, they can have vulnerabilities and become exploited.

This thesis focuses on transaction order dependence (TOD), which is a prerequisite for a kind of attack called front-running. TOD means that the state changes performed by transactions depend on the order in which the transactions are executed. In a front-running attack, an attacker sees that someone is about to perform a transaction and then quickly inserts a transaction before it. Because of TOD, executing the attacker's transaction before the victim's transaction yields different state changes than when execution the victim's transaction first.

This work proposes a method to take a pair of transactions and to simulate the two transactions in both orders. When executing the transactions in both orders, we can compare their behaviors to see if they are TOD and also if it exhibits characteristics of a front-running attack.

We use the state changes of transactions to calculate the world states used for transaction execution. This removes the need to execute intermediary transactions that were originally executed between the two transactions we analyze. Instead, we can use their state changes to maintain the effect they had on the second transaction, while updating the world states according to the different orders.

Moreover, we search the blockchain history for transaction pairs that are potentially TOD. We match transactions that access and modify the same state and define several filters to remove irrelevant transaction pairs. On these transaction pairs, we use our simulation method to check if they are TOD and if they have characteristics of a front-running attack. We check for the characteristics of the ERC-20 multiple withdrawal attack [1], the TOD properties implemented by Securify [2], and financial gains and losses[3].

We show that our concepts can be implemented with endpoints exposed by an archive node. We neither require custom modifications nor local access to an archive node.

Overall, our main contributions are:

- A method to simulate a pair of transactions in two different orders.
- A precise definition of TOD in the context of blockchain transaction analysis.
- An evaluation of an approximation for TOD.
- A compilation of EVM instructions that can cause TOD.

- A method to mine and filter transaction pairs that are potentially TOD.

1.1 Related works

The studies by W. Zhang *et al.* [3] and C. F. Torres, R. Camino, and R. State [4] both detect and analyze front-running attacks that occurred on the Ethereum blockchain. We discuss potential inaccuracies in their simulation approaches. Contrary to these works, our study focuses on TOD, a prerequisite of front-running. However, we also implement the attack definition by W. Zhang *et al.* [3] to compare our results with theirs.

P. Daian *et al.* [5] detect a specific kind of front-running attack by observing transaction executions. They measure so-called arbitrage opportunities, where a single transaction can make net revenue. While this is TOD, as only the first transaction that uses an arbitrage opportunity makes revenue, they do not need to simulate the different transaction orders for their analysis. Similarly, Y. Wang, P. Zuest, Y. Yao, Z. Lu, and R. Wattenhofer [6] also study a type of front-running attack without simulating different transaction orders.

D. Perez and B. Livshits [7] explicitly analyze transactions for TOD. They do so by recording for each transaction which storage it accessed and modified and then matching transactions where these overlap. Our work discusses the theoretical background of this approach and our method to detect potential TODs works in a similar way as theirs.

Several other works provide frameworks to analyze attack transactions in Ethereum [8]–[11]. None of these frameworks supports the simulation of transactions in different orders, therefore we cannot directly use them to detect TOD. Regarding the use of archive nodes, an evaluation by S. Wu *et al.* [10] states that replaying transactions with them is slow, taking "[...] more than 47 min to replay 100 normal transactions". However, C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State [8] show that it is indeed feasible to use archive nodes for attack analysis. We follow this work and use archive nodes to implement our simulation method.

2 Background

This chapter provides background knowledge on Ethereum that is helpful for following the remaining paper. We also introduce a notation for these concepts.

2.1 Ethereum

Ethereum is a blockchain that can be characterized as a "transactional singleton machine with shared-state" [12, p.1]. By using a consensus protocol, a decentralized set of nodes agrees on a globally shared state. This state contains two types of accounts: *externally owned accounts* (EOA) and *contract accounts* (also referred to as smart contracts). The shared state is modified by executing *transactions* [13].

2.2 World State

Similar to [12, p.3], we will refer to the shared state as *world state*. The world state maps each 20-byte address to an account state, containing a *nonce*, *balance*, *storage* and *code*[1]. They store the following data [12, p.4]:

- *nonce*: For EOAs, this is the number of transactions submitted by this account. For contract accounts, this is the number of contracts created by this account.
- *balance*: The value of Wei this account owns, the smallest unit of Ether.
- *storage*: The storage allows contract accounts to persist information across transactions. It is a key-value mapping where both, key and value, are 256 bits long. For EOAs, the storage is empty.
- *code*: For contract accounts, the code is a sequence of EVM instructions.

We denote the world state as σ and the value at a specific *state key* K as $\sigma(K)$. For the nonce, balance and code the state key denotes the state type and the account's address, written as $\sigma((\text{'nonce'}, a))$, $\sigma((\text{'balance'}, a))$ and $\sigma((\text{'code'}, a))$, respectively. For the value at a storage slot k we use $\sigma((\text{'storage'}, a, k))$.

2.3 EVM

The Ethereum Virtual Machine (EVM) is used to execute code in Ethereum. It executes instructions that can access and modify the world state. The EVM is Turing-complete, except that it is executed with a limited amount of *gas*, and each instruction costs some

[1]Technically, the account state only contains hashes that identify the storage and code, not the actual storage and code. This distinction is not relevant in this paper, therefore we simply refer to them as storage and code.

gas. When it runs out of gas, the execution will halt [12, p.14]. This prevents infinite loops, as their execution exceeds the gas limit.

Most EVM instructions are formally defined in the Yellowpaper [12, p.30-38]. However, the Yellowpaper currently does not include the changes from the Cancun upgrade [14], therefore we will also refer to the informal descriptions available on evm.codes [15].

2.4 Transactions

A transaction can modify the world state by transferring Ether and executing EVM code. It must be signed by the owner of an EOA and contains the following data relevant to our work:

- *sender*: The address of the EOA that signed this transaction.[2]
- *recipient*: The destination address.
- *value*: The value of Wei that should be transferred from the sender to the recipient.
- *gasLimit*: The maximum amount of gas that can be used for the execution.

If the recipient address is empty, the transaction will create a new contract account. These transactions also include an *init* field that contains the code to initialize the new contract account.

When the recipient address is given and a value is specified, this will be transferred to the recipient. Moreover, if the recipient is a contract account, it also executes the recipient's code. The transaction can specify a *data* field to pass input data to the code execution [12, p.4-5].

For every transaction, the sender must pay a *transaction fee*. This fee is composed of a *base fee* and a *priority fee*. Every transaction must pay the base fee. The amount of Wei will be withdrawn from the sender and not given to any other account. For the priority fee, the transaction can specify if and how much they are willing to pay. This fee will be taken from the sender and given to the block validator, which is explained in the next section [12, p.8].

We denote a transaction as T, sometimes adding a subscript T_A to differentiate it from another transaction T_B.

[2]The sender is implicitly given through a valid signature and the transaction hash [12, p.25-27]. We are only interested in transactions that are included in the blockchain, thus the signature must be valid, and the transaction's sender can always be derived.

2.5 Blocks

The Ethereum blockchain consists of a sequence of blocks, where each block builds upon the state of the previous block. To achieve consensus about the canonical sequence of blocks in a decentralized network of nodes, Ethereum uses a consensus protocol. In this protocol, validators build and propose blocks to be added to the blockchain [16]. It is the choice of the validator which transactions to include in a block. However, they are incentivized to include transactions that pay high transaction fees, as they receive the fee [12, p.8].

Each block consists of a block header and a sequence of transactions that are executed in this block.

2.6 Transaction submission

This section discusses how a transaction signed by an EOA ends up being included in the blockchain.

Traditionally, the signed transaction is broadcasted to the network of nodes, which temporarily store them in a *mempool*, a collection of pending transactions. The current block validator then picks transactions from the mempool and includes them in the next block. With this submission method, the pending transactions in the mempool are publicly known to the nodes in the network, even before being included in the blockchain. This time window will be important for our discussion on front-running, as it gives nodes time to react to a transaction before it becomes part of the blockchain [17].

A different approach, the Proposer-Builder Separation (PBS), has gained more popularity recently. Here, we separate the task of collecting transactions and building blocks with them from the task of proposing them as a validator. A user submits their signed transaction or transaction bundle to a block builder. The block builder has a private mempool and uses it to create profitable blocks. Finally, the validator picks one of the created blocks and adds it to the blockchain [18].

2.7 Transaction execution

In Ethereum, transaction execution is deterministic [12, p.9]. Transactions can access the world state and their block environment, therefore their execution can depend on these values. After executing a transaction, the world state is updated accordingly.

We denote a transaction execution as $\sigma \xrightarrow{T} \sigma'$, implicitly letting the block environment correspond to the transaction's block. Furthermore, we denote the state change by a transaction T as Δ_T, with prestate$(\Delta_T) = \sigma$ being the world state before execution and poststate$(\Delta_T) = \sigma'$ the world state after the execution of T.

For two state changes Δ_{T_A} and Δ_{T_B}, we say that they are equivalent, $\Delta_{T_A} \sim \Delta_{T_B}$, if the relative change of the values is equal. Formally, let $\Delta_{T_A} \sim \Delta_{T_B}$ be true if and only if:

$$\forall K : \text{poststate}\left(\Delta_{T_A}\right)(K) - \text{prestate}\left(\Delta_{T_A}\right)(K) = \text{poststate}\left(\Delta_{T_B}\right)(K) - \text{prestate}\left(\Delta_{T_B}\right)(K)$$

We extend this equivalence definition to sequences of state changes by summing up the differences of the state changes on both sides. We define that two sequences of state changes $\langle \Delta_{T_{A_0}}, ..., \Delta_{T_{A_n}} \rangle$ and $\langle \Delta_{T_{B_0}}, ..., \Delta_{T_{B_m}} \rangle$ are equivalent if:

$$\forall K : \sum_{i=0}^{n} \text{poststate}\left(\Delta_{T_{A_i}}\right)(K) - \text{prestate}\left(\Delta_{T_{A_i}}\right)(K) = \sum_{j=0}^{m} \text{poststate}\left(\Delta_{T_{B_j}}\right)(K) - \text{prestate}\left(\Delta_{T_{B_j}}\right)(K)$$

For example, if both Δ_{T_A} and Δ_{T_B} increase the balance at address a by 10 Wei and make no other state changes, then $\Delta_{T_A} \sim \Delta_{T_B}$. If one of them had modified it by e.g. 15 Wei or 0 Wei, or additionally modified some storage slot, we would have $\Delta_{T_A} \nsim \Delta_{T_B}$.

We define $\sigma + \Delta_T$ to be equal to the state σ, except that every state that was changed by the execution of T is overwritten with the value in poststate(Δ_T). Similarly, $\sigma - \Delta_T$ is equal to the state σ, except that every state that was changed by the execution of T is overwritten with the value in prestate(Δ_T). Formally, these definitions are as follows:

$$\text{changed\_keys}(\Delta_T) := \{K \mid \text{prestate}(\Delta_T)(K) \neq \text{poststate}(\Delta_T)(K)\}$$

$$(\sigma + \Delta_T)(K) := \begin{cases} \text{poststate}(\Delta_T)(K) & \text{if } K \in \text{changed\_keys}(\Delta_T) \\ \sigma(K) & \text{otherwise} \end{cases}$$

$$(\sigma - \Delta_T)(K) := \begin{cases} \text{prestate}(\Delta_T)(K) & \text{if } K \in \text{changed\_keys}(\Delta_T) \\ \sigma(K) & \text{otherwise} \end{cases}$$

For instance, if transaction T changed the storage slot 1234 at address 0xabcd from 0 to 100, then we have changed_keys$(\Delta_T) = \{(\text{'storage'}, \text{0xabcd}, 1234)\}$. Further, we have $(\sigma + \Delta_T)((\text{'storage'}, \text{0xabcd}, 1234)) = 100$ and $(\sigma - \Delta_T)((\text{'storage'}, \text{0xabcd}, 1234)) = 0$. For all other storage slots k we have $(\sigma + \Delta_T)((\text{'storage'}, a, k)) = \sigma((\text{'storage'}, a, k)) = (\sigma - \Delta_T)((\text{'storage'}, a, k))$.

2.8 Nodes

A node consists of an *execution client* and a *consensus client*. The execution client keeps track of the world state and the mempool and executes transactions. The consensus client takes part in the consensus protocol. For this work, we will use an *archive node*, which is a node that allows reproducing the state and transactions at any block [19].

2.9 RPC

Execution clients implement the Ethereum JSON-RPC specification [20]. This API gives remote access to an execution client, for instance, to inspect the current block number with `eth_blockNumber` or to execute a transaction without committing the state via `eth_call`. In addition to the standardized RPC methods, we will also make use of methods in the debug namespace, such as `debug_traceBlockByNumber`. While this namespace is not standardized, several execution clients implement these additional methods [21]–[23].

2.10 Tokens

In Ethereum, tokens are assets managed by contract accounts [24]. The contract account stores which address holds how many tokens. There are several token standards that a contract account can implement, allowing standardized methods to interact with the token. For instance, the ERC-20 standard defines a `transfer` method, which allows the holder of a token to transfer the token to someone else [25].

3 Transaction order dependency

In this chapter we discuss our definitions of transaction order simulations and transaction order dependency (TOD). We first introduce the idea of TOD with a preliminary definition and then show several shortcomings of this simple definition. Based on these insights, we construct precise definitions of TOD and our transaction order simulation.

3.1 Approaching TOD

Intuitively, a pair of transactions (T_A, T_B) is transaction order dependent (TOD), if the result of sequentially executing the transactions depends on the order of their execution. As a preliminary TOD definition we use the following:

$$\sigma \xrightarrow{T_A} \sigma_1 \xrightarrow{T_B} \sigma'$$

$$\sigma \xrightarrow{T_B} \sigma_2 \xrightarrow{T_A} \sigma''$$

$$\sigma' \neq \sigma''$$

So, starting from an initial state, when we execute first T_A and then T_B it will result in a different state, than when we execute T_B and afterwards T_A.

We will refer to the execution order $T_A \to T_B$, the one that occurred on the blockchain, as the *normal* execution order, and $T_B \to T_A$ as the *reverse* execution order.

3.2 Motivating examples

We provide two examples to illustrate how TOD can be exploited.

3.2.1 Password leaking

The first example is an attack from a dataset by [4][3]. A simplified version of the vulnerable contract with added comments is presented below. It allows depositing some Ether and locking it with a password, and then anyone with the password can withdraw this Ether.

[3]The attacker transaction is 0x15c0d7252fa93c781c966a98ab46a1c8c086ca2a0da7eb0a7a06c522818757da, and the victim transaction is 0x282e4de019b59a50b89c1fdc2e70c4bbd45a7ad7f7a1a6d4807a587b5fcdcdf6.

```
contract PasswordEscrow {
  struct Transfer {
    address from;
    uint256 amount;
  }

  mapping(bytes32 => Transfer) private transferToPassword;

  function deposit(bytes32 _password) public payable {
    // REMARK: this stores an entry for the password and saves the amount
of Ether
    // that was sent along the transaction
    bytes32 pass = sha3(_password);
    transferToPassword[pass] = Transfer(msg.sender, msg.value);
  }

  function getTransfer(bytes32 _password) public payable {
    // REMARK: this verifies that an entry for the password exists
    // and gets the amount of Ether that was deposited for the password
    require(
      transferToPassword[sha3(_password)].amount > 0
    );
    bytes32 pass = sha3(_password);
    uint256 amount = transferToPassword[pass].amount;

    transferToPassword[pass].amount = 0;

    // REMARK: this transfers the Ether to the account that transaction's
sender
    msg.sender.transfer(amount);
  }
}
```

Contract 1: The `PasswordEscrow` contract. This is a simplified version with added comments, based on the source code from Etherscan [26].

The victim previously interacted with the contract to deposit some Ether and lock it with a password. For the sake of the argument, we ignore that the password is already public at this step. This could be fixed, e.g. by directly submitting `sha3(password)` rather than the password itself, without resolving the TOD issue we discuss here.

Later, the victim tried to withdraw this Ether by creating a transaction that calls `getTransfer` with the password. However, in the time between the transaction submission and its inclusion in a block, an attacker saw this transaction and determined that they can perform the Ether withdrawal themselves. They copied the transaction data, including the password, and submitted their own transaction with a higher gas price than the victim's. The attacker's transaction ended up being executed first and withdrew all the Ether.

If we map this attack to our preliminary TOD definition above, the first transaction that executes will withdraw the Ether and thus increase the sender's balance. If the attacker's transaction executes first, we end up in a state where the attacker has more balance than if the victim's transaction is executed first. Therefore, $\sigma' \neq \sigma''$.

3.2.2 ERC-20 multiple withdrawal

As a second example, we explain the ERC-20 multiple withdrawal attack [1]. Contracts that implement the ERC-20 token standard must include an `approve` method [25]. This method takes as parameters a `spender` and a `value` and allows the `spender` to spend `<value>` tokens from your account. For instance, when some account a calls `approve(b, 0x1234)`, then `b` can transfer `0x1234` tokens from a to any other account. If the `approve` method is called another time, the currently approved value is overwritten with the new value, regardless of the previous value.

We illustrate that approvals and the spending of approved tokens can be TOD in Table 1. In the benign scenario, b spends one token and remains with two tokens that are still approved. However, in the attack scenario, b spends one token and only afterwards a approves b to spend three tokens. Therefore, b remains with three tokens approved tokens instead of two. As such, changing the order of the second and third transaction results in different states, hence they are TOD.

From the perspective of a, they only wanted to allow b to use three tokens. However, when b reacts to a pending approval by executing a `transferFrom` before the approval is included in a block, then b is able to use more than three tokens in total. This happened in the attack scenario, where the `transferFrom` is executed before the second `approve` got included in a block.

Benign scenario

Action	Approved tokens
`approve(b, 1)`	1
`approve(b, 3)`	3
`transferFrom(a, b, 1)`	2

Attack scenario

Action	Approved tokens
`approve(b, 1)`	1
`transferFrom(a, b, 1)`	0
`approve(b, 3)`	3

Table 1: Benign and attack scenario for ERC-20 approvals.

3.3 Relation to previous works

This section discusses, how our preliminary TOD definition relates to previous works that detect front-running attacks.

In [4], the authors do not provide a formal definition of TOD or front-running attacks. Nevertheless, for displacement attacks, they include the following check to detect if two transactions fall into this category:

> "[...] we run in a simulated environment first T_A before T_V and then T_V before T_A. We report a finding if the number of executed EVM instructions is different across both runs for T_A and T_V, as this means that T_A and T_V influence each other."

Similar to our preliminary TOD definition, they execute T_A and T_V in different orders and check if it affects the result. In their case, they only check the number of executed instructions, instead of the resulting state. This check misses attacks where the same instructions are executed, but the operands of instructions in the second transaction change because of the first transaction.

In [3], the authors define an attack as a triple $A = \langle T_a, T_v, T_a^p \rangle$, where T_a and T_v are similar to T_A and T_B from our definition, and T_a^p is an optional third transaction. They consider the execution orders $T_a \to T_v \to T_a^p$ and $T_v \to T_a \to T_a^p$ and check if the execution order influences financial gains, which we will discuss in more detail in Section 6.1.

We note that if these two execution orders result in different states, this is not because of the last transaction T_a^p, but because of a TOD between T_a and T_v. As we always execute T_a^p last, and transaction execution is deterministic, it only gives a different result if the execution of T_a and T_v gave a different result. Therefore, if the execution order results in different financial gains, then T_a and T_v must be TOD.

3.4 Imprecise definitions

Our preliminary definition of TOD, and the related definitions above, are not precise regarding the semantics of a reordering of transactions and their executions. This makes it impossible to apply exactly the same methodology without analyzing the source code related to the papers. We describe three issues where the definition is not precise enough, and show how these are differently interpreted by the two papers.

For the analysis of the tools by [3] and [4], we will use the current version of the source codes, [27] and [28], respectively.

3.4.1 Intermediary transactions

To analyze a TOD (T_A, T_B), we are interested in how T_A affects T_B in the normal order, and how T_B affects T_A in the reverse order. Our preliminary definition does not specify how to handle transactions that occur between T_A and T_B, which we will name *intermediary transactions.*

Suppose that there is one transaction T_X between T_A and T_B: $\sigma \xrightarrow{T_A} \sigma_A \xrightarrow{T_X} \sigma_{AX} \xrightarrow{T_B} \sigma_{AXB}$. The execution of T_B may depend on both T_A and T_X. When we are interested in the effect of T_A on T_B, we need to define what happens with T_X.

For executing in the normal order, we have two possibilities:

1. $\sigma \xrightarrow{T_A} \sigma_A \xrightarrow{T_X} \sigma_{AX} \xrightarrow{T_B} \sigma_{AXB}$, the same execution as on the blockchain, including the effects of T_X.
2. $\sigma \xrightarrow{T_A} \sigma_A \xrightarrow{T_B} \sigma_{AB}$, leaving out T_X and thus having a normal execution that potentially diverges from the results on the blockchain (as σ_{AB} may differ from σ_{AXB}).

When executing the reverse order, we have the following choices:

1. $\sigma \xrightarrow{T_B} \sigma_B \xrightarrow{T_A} \sigma_{BA}$, which ignores T_X and thus may influence the execution of T_B.
2. $\sigma \xrightarrow{T_X} \sigma_X \xrightarrow{T_B} \sigma_{XB} \xrightarrow{T_A} \sigma_{XBA}$, which executes T_X on σ rather than σ_A and now also includes the effects of T_X for executing T_A.
3. $\sigma \xrightarrow{T_B} \sigma_B \xrightarrow{T_X} \sigma_{BX} \xrightarrow{T_A} \sigma_{BXA}$, which executes T_X after T_B and before T_A, thus potentially influencing the execution of both T_A and T_B.

All of these scenarios are possible, but none of them provides a clean solution to solely analyze the effect of T_A on T_B, as we always may have some indirect effect from the (non-)execution of T_X.

In [3], this influence of intermediary transactions is acknowledged as causing a few false positives:

> "In blockchain history, there could be many other transactions between T_a, T_v, and T_p^a. When we change the transaction orders to mimic attack-free scenarios, the relative orders between T_a (or T_v) and other transactions are also changed. Financial profits of the attack or victim could be affected by such relative orders. As a result, the financial profits in the attack-free scenario could be incorrectly calculated, and false-positively reported attacks may be induced, but our manual check shows that such cases are rare."

Nonetheless, it is not clear, which of the above scenarios they applied for their analysis. The other work, [4], does not mention this issue of intermediary transactions.

3.4.1.a Code analysis of [3]

In [3], algorithm 1 takes all the executed transactions as its input. These transactions and their results are used in the `searchVictimGivenAttack` method, where `ar` represents the attack transaction with result and `vr` represents the victim transaction with result.

For the normal execution order ($T_a \rightarrow T_v$), the authors use `ar` and `vr` and pass them to their `CheckOracle` method, which then compares the resulting states. As `ar` and `vr` are obtained by executing all transactions, they also include the intermediary transactions for these results (similar to our $\sigma \xrightarrow{T_A} \sigma_A \xrightarrow{T_X} \sigma_{AX} \xrightarrow{T_B} \sigma_{AXB}$ case).

For the reverse order ($T_v \rightarrow T_a$), they take the state before T_a, i.e. σ. Then they execute all transactions obtained from the `SlicePrerequisites` method. And finally, they execute T_v and T_a.

The `SlicePrerequisites` method uses the `hbGraph`, which is built in `StartSession`. `hbGraph` seems to be a graph where each transaction points to the previous transaction from the same EOA. The `SlicePrerequisites` method uses this graph to obtain all transactions between T_a and T_v that are from the same sender as T_v. This interpretation matches the test case "should slide prerequisites correctly" from the source code. As the paper does not mention these prerequisite transactions, we do not know why this subset of intermediary transactions was chosen.

We can conclude that [3] executes all intermediary transactions in the normal order. However, in the reverse order, they only execute intermediary transactions that are also sent by the victim, but do not execute any other intermediary transactions.

3.4.1.b Code analysis of [4]

In the file `displacement.py`, lines 154-155 replay the normal execution order, and lines 158-159 the reverse execution order. They only execute T_A and T_V (in normal and reverse order), but do not execute any intermediate transactions.

3.4.2 Block environments

When we analyze a pair of transactions (T_A, T_B), it may happen that these are not part of the same block. The execution of the transactions may depend on the block environment they are executed in, for instance, if they access the current block number. Thus, executing T_A or T_B in a block environment different from the blockchain may alter their behavior. From our preliminary TOD definition, it is not clear which block environment(s) we use when replaying the transactions in normal and reverse order.

3.4.2.a Code analysis of [3]

In the normal scenario, the block environments used are the same as originally used for the transaction.

For the reverse scenario, the block environment used to execute all transactions is contained in `ar.VmContext` and corresponds to the block environment of T_a. Therefore, T_a is executed in the same block environment as on the blockchain, while T_v and the intermediary transactions may be executed in a block environment different from the normal scenario.

3.4.2.b Code analysis of [4]

In the file `displacement.py` line 151, we see that the emulator uses the same block environment for both transactions. Therefore, at least one of them will be executed in a block environment different from the blockchain. However, it uses the same block environment for both scenarios, thus being consistently different from the execution on the blockchain.

3.4.3 Initial state σ

While our preliminary TOD definition specifies that we start with the same σ in both execution orders, it is up to interpretation which world state σ actually designates.

3.4.3.a Code analysis of [3]

Both, in the normal and reverse scenario, it uses the state directly before executing T_a, including the state changes of previous transactions within the same block. In the reverse scenario, this is the case as it uses `ar.State`.

3.4.3.b Code analysis of [4]

The emulator is initialized with the block `front_runner["blockNumber"]-1` and no single transactions are executed prior to running the analysis. Therefore, the state cannot include transactions that were executed in the same block before T_A.

Similar to the case with the block environment, this could lead to differences between the emulation and the results from the blockchain when T_A or T_V are affected by a previous transaction in the same block.

3.5 TOD simulation

To address the issues above, we define a TOD simulation method that explicitly defines the used world states and block environments while also taking intermediary transactions into account:

Definition 3.5.1 (Normal and reverse scenarios): Consider a sequence of transactions, with σ being the world state right before T_A was executed on the blockchain:

$$\sigma \xrightarrow{T_A} \sigma_A \xrightarrow{T_{X_1}} \dots \xrightarrow{T_{X_n}} \sigma_{X_n} \xrightarrow{T_B} \sigma_B$$

Let Δ_{T_A} and Δ_{T_B} be the corresponding state changes from executing T_A and T_B, and let all transactions be executed in the same block environment as they were executed on the blockchain.

Let Δ'_{T_B} be the state change when executing $\left(\sigma_{X_n} - \Delta_{T_A}\right) \xrightarrow{T_B} \sigma'_B$ and Δ'_{T_A} be the state change when executing $\left(\sigma + \Delta'_{T_B}\right) \xrightarrow{T_A} \sigma'_A$.

We call Δ_{T_A} and Δ_{T_B} the state changes from the normal scenario and Δ'_{T_A} and Δ'_{T_B} the state changes from the reverse scenario.

The normal scenario represents the order $T_A \to T_B$. The state changes Δ_{T_A} and Δ_{T_B} are equal to the ones observed on the blockchain, as we execute the transactions in their original block environment and with their original prestate.

The reverse scenario models the order $T_B \to T_A$, where T_B occurs before T_A. Therefore, we execute T_B on a state that does not contain the changes of T_A. We do so by taking the world state exactly before executing T_B, namely σ_{X_n}, and then removing the state changes of T_A by computing $\sigma_{X_n} - \Delta_{T_A}$. Executing T_B on $\sigma_{X_n} - \Delta_{T_A}$ gives us the state change Δ'_{T_B}. To model the execution of T_A after T_B, we take the state σ on which T_A was originally executed and add the state changes Δ'_{T_B}.

3.6 TOD definition

Based on the definition of normal and reverse scenarios, we define TOD as follows:

Definition 3.6.1 (TOD): Let T_A and T_B be two transactions with the corresponding state changes Δ_{T_A} and Δ_{T_B} from the normal scenario and Δ'_{T_A} and Δ'_{T_B} from the reverse scenario.

We say that (T_A, T_B) is TOD if and only if $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$.

Consider the example of the ERC-20 multiple withdrawal from Section 3.2.2, with T_A being the attacker transaction that calls `transferFrom(a, b, 1)` and T_B being the victim transaction that calls `approve(b, 3)`. In the normal scenario, we have shown that the attacker remains with three approved tokens, while in the reverse scenario, only two tokens would remain. Intuitively, this satisfies $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$, as the change approved tokens differs between the normal and the reverse scenario.

More formally, let K be the state key that tracks how many tokens are approved by a for b. Initially, one token is approved, therefore $\sigma(K) = 1$. When executing T_A in the normal scenario, where the attacker spends the one approved token, this changes to $\sigma(K) = 0$. Therefore, we have a change of $\text{poststate}\left(\Delta_{T_A}\right)(K) - \text{prestate}\left(\Delta_{T_A}\right)(K) = -1$. We then continue to execute T_B in the normal scenario, which sets $\sigma(K) = 3$, therefore $\text{poststate}\left(\Delta_{T_B}\right)(K) - \text{prestate}\left(\Delta_{T_B}\right)(K) = 3$. When we add up these two state changes, we get an overall state change of 2 for the state at key K. However, doing the same calculations for the reverse scenario results in an overall state change of 1 for K, as T_B first increases it by two and T_A then reduces it by one. As the overall changes differ between the normal and reverse scenario, we have $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$ and (T_A, T_B) is TOD.

Similarly, for the password leaking example in Section 3.2.1 we showed that the execution order determines who can withdraw the stored Ether. If the attacker's transaction is executed first, they withdraw the Ether. If it is executed second, the attacker does not withdraw any Ether. Therefore, the change at the state key $K = (\text{'balance'}, \textit{attacker})$ depends on the transaction order, and thus, the transactions are TOD.

3.7 TOD approximation

This paper focuses on detecting TOD attacks, in which the attacker inserts a transaction prior to some transaction T. We assume that the first transaction tries to influence the second transaction, which implies that in every TOD attack, the state changes of T_B

should be dependent on the transaction order. We use this assumption to define an approximation of TOD:

Definition 3.7.1 (Approximately TOD): Let T_A and T_B be transactions with the state changes Δ_{T_B} for the normal scenario and Δ'_{T_B} for the reverse scenario.

We say that (T_A, T_B) is approximately TOD if and only if $\Delta_{T_B} \not\approx \Delta'_{T_B}$.

In principle, the assumption that an attack influences the transaction it front-runs need not hold. For example, suppose a transaction T leaks a password that can be used to withdraw Ether, but at the same time, T locks the contract that contains this Ether. An attacker may use the password to withdraw the Ether without necessarily influencing the execution of T but it needs need to front-run T because of the contract locking.

3.8 Definition strengths

3.8.1 Performance

To check if two transactions, T_A and T_B, are TOD, we need the initial world state σ, and the state changes from T_A, T_B and the intermediary transactions T_{X_n}. With the state changes we can compute $\sigma_{X_n} - \Delta_{T_A} = \sigma + \Delta_{T_A} + \left(\sum_{i=0}^{n} \Delta_{T_{X_i}}\right) - \Delta_{T_A}$ and then execute T_B on this state. With the recorded state changes, Δ'_{T_B}, we can compute $\sigma + \Delta'_{T_B}$ and execute T_A on this state. As such, we need one transaction execution to check for the TOD approximation and two transaction executions to check for TOD. Despite including the effect of arbitrarily many intermediary transactions, we do not need to execute them to check for TOD.

When we want to check n transactions for TOD, there are $\frac{n^2-n}{2}$ possible transaction pairs. Thus, if we want to test each pair for TOD we end up with a total of $\frac{n^2-n}{2}$ transaction executions for the approximation and $n^2 - n$ executions for the exact TOD check. Similar to [4] and [3], we can filter irrelevant transaction pairs to reduce the search space.

Depending on the available world states and state changes, the exact number of required transaction executions and the method to compute world states may differ. For instance, the archive nodes Erigon 2 and Reth currently only store state changes for each block, but not on a transaction level [29], [30]. We show the state calculations under such constraints in Section 5. Other systems, such as EthScope [10], and Erigon 3 [31], store changes for every transaction. However, EthScope is not publicly available anymore and Erigon 3 is still in development.

3.8.2 Similarity to blockchain executions

With our definition, the state changes Δ_{T_A} and Δ_{T_B} from the normal execution are equivalent to the state changes that happened on the blockchain. Also, the reverse order is closely related to the state from the blockchain, as we start with the world states before T_A and T_B and only change state keys that were modified by T_A and T_B, thus only the state keys relevant for TOD simulation. Furthermore, we prevent the effects of block environment changes by using the same environments as on the blockchain.

This contrasts with other implementations, where transactions are executed in different block environments than originally, a different world state is used for the first transaction or the effect of intermediary transactions is ignored. All three cases can alter the execution of T_A and T_B, such that the result is not closely related to the blockchain.

3.9 Definition weaknesses

3.9.1 Approximation focuses on effect on T_B

In some cases, the transaction order can affect the execution of the individual transactions, but does not affect the overall result of executing both transactions. The approximation does not consider the execution of T_A after T_B in the reverse order, which could lead to incorrect TOD classification.

For example, consider the case where both T_A and T_B multiply a value in a storage slot by 5. If the storage slot initially has the value 1, then executing both T_A and T_B will result in 25, regardless of the order. However, the state changes Δ_{T_B} and Δ'_{T_B} are different, as for one scenario, the value changes from 1 to 5 and for the other from 5 to 25. Therefore, this would be classified as approximately TOD.

Note that the approximation is robust against the cases, where the absolute values differ, but the change is constant. For instance, if both T_A and T_B would increase the storage slot by 5 rather than multiplying it, the state changes Δ_{T_B} and Δ'_{T_B} would be from 1 to 6 and from 6 to 11. As our definition for state change equivalence uses the difference between the state before and after execution, we would compare the change $6 - 1 = 5$ against $11 - 6 = 5$, thus $\Delta_{T_B} \sim \Delta'_{T_B}$.

3.9.2 Indirect dependencies

An intuitive interpretation of our TOD definition is that we compare $T_A \to T_{X_i} \to T_B$ with $T_{X_i} \to T_B \to T_A$, i.e. reckon what happens if T_A is not executed first but last. However, the definition we provide does not perfectly match this concept, because it does not consider interactions between T_A and the intermediary transactions T_{X_i}. In the intuitive model, not executing T_A before the intermediary transactions may influence them and thus indirectly change the behavior of T_B. Then, we do not know if T_A directly influences T_B, or only through some interplay with intermediary transactions. Similarly, when executing T_A last, we do not know if T_A behaves differently this is because of an interaction with T_B or an intermediary transaction.

Therefore, our exclusion of interactions between T_A and T_{X_i} may be desirable to focus only on interactions between T_A and T_B, however it can cause divergences between our analysis results and what would have happened on the blockchain.

As an example, consider the three transactions T_A, T_X and T_B:

1. T_A: sender a transfers 5 Ether to address x.
2. T_X: sender x transfers 5 Ether to address b.
3. T_B: sender b transfers 5 Ether to address y.

When executing these transactions in the normal order, and a initially has 5 Ether and the others have 0, then all of these transactions succeed. If we remove T_A and only execute T_X and T_B, then firstly T_X would fail, as x did not get the 5 Ether from a, and consequently, also T_B fails.

However, when using our TOD definition and computing $\left(\sigma_{X_n} - \Delta_{T_A}\right)$, we would only modify the balances for a and x, but not for b, because b is not modified in Δ_{T_A}. Thus, T_B would still succeed in the reverse order according to our definition, but would fail in practice due to the indirect effect. This shows, how the concept of removing T_A does not map exactly to our TOD definition.

In this example, we had a TOD for (T_A, T_X) and (T_X, T_B). However, we can also have an indirect dependency between T_A and T_B without a TOD for (T_X, T_B). For instance, if T_X and T_B would be TOD, but T_A caused T_X to fail. When inspecting the normal order, T_X failed, so there is no TOD between T_X and T_B. However, when executing the reverse order without T_A, then T_X would succeed and could influence T_B.

3.10 State collisions

We denote the state accesses by a transaction T as a set of state keys $R_T = \{K_1, ..., K_n\}$ and the state modifications as $W_T = \{K_1, ..., K_m\}$.

Inspired by the definition of a transaction race in [32], we define the state collisions of two transactions as:

$$\text{collisions}(T_A, T_B) = \left(W_{T_A} \cap R_{T_B}\right) \cup \left(W_{T_A} \cap W_{T_B}\right)$$

For instance, if transaction T_A modifies the balance of some address a, and T_B accesses this balance, we have $\text{collisions}(T_A, T_B) = (\{(\text{‘balance’}, a)\} \cap \{(\text{‘balance’}, a)\}) \cup (\{(\text{‘balance’}, a)\} \cap \emptyset) = \{(\text{‘balance’}, a)\}$.

With $W_{T_A} \cap R_{T_B}$ we include write-read collisions, where T_A modifies some state key and T_B accesses the same state key. With $W_{T_A} \cap W_{T_B}$ we include write-write collisions, where both transactions write to the same state location, for instance by writing to the same storage. Following the assumption of the TOD approximation, we do not include $R_{T_A} \cap W_{T_B}$, as in this case T_A does not influence the execution of T_B.

3.11 TOD candidates

We will refer to a transaction pair (T_A, T_B), where T_A was executed before T_B and $\text{collisions}(T_A, T_B) \neq \emptyset$ as a TOD candidate.

A TOD candidate is not necessarily TOD or approximately TOD. For instance, consider the case that T_B only reads the value that T_A wrote but never uses it for any computation. This would be a TOD candidate, as they have a collision, however the result of executing T_B is not influenced by this collision.

If (T_A, T_B) is approximately TOD, then (T_A, T_B) must also be a TOD candidate. We can only have $\Delta_{T_B} \nsim \Delta'_{T_B}$ if the state keys that T_B accesses or modifies differ between the normal and reverse scenarios. As the only difference between the scenarios is the removal of Δ_{T_A} in the reverse scenario, the differences of the state keys must come from Δ_{T_A}. Therefore, T_A also modifies these state keys, and we have $\left(W_{T_A} \cap R_{T_B}\right) \cup \left(W_{T_A} \cap W_{T_B}\right) \neq \emptyset$. This is equivalent to $\text{collisions}(T_A, T_B) \neq \emptyset$, showing that (T_A, T_B) must be a TOD candidate.

Therefore, the set of all approximately TOD transaction pairs is a subset of all TOD candidates.

In the case that (T_A, T_B) is TOD but not approximately TOD, the pair (T_A, T_B) need not be a TOD candidate. By the definitions of TOD and approximately TOD, we have $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$ and $\Delta_{T_B} \sim \Delta'_{T_B}$, which implies that $\Delta_{T_A} \nsim \Delta'_{T_A}$ must hold. Similar to the previous argument, $\Delta_{T_A} \nsim \Delta'_{T_A}$ implies $\left(R_{T_A} \cap W_{T_B}\right) \cup \left(W_{T_A} \cap W_{T_B}\right) \neq \emptyset$. However, in this case we cannot conclude $\text{collisions}(T_A, T_B) \neq \emptyset$, because we excluded $R_{T_A} \cap W_{T_A}$ from our collision definition.

As such, the definition of TOD candidates aligns with the approximation of TOD, but not necessarily the exact TOD definition.

3.12 Causes of state collisions

This section discusses what can cause two transactions T_A and T_B to have state collisions. To do so, we show the ways a transaction can access and modify the world state.

3.12.1 Causes with code execution

When the recipient of a transaction is a contract account, it will execute the recipient's code. The code execution can access and modify the state through several instructions. By inspecting the EVM instruction definitions [12, p.30-38], [15], we compiled a list of instructions that can access and modify the world state.

In Table 2, we see the instructions that can access the world state. For most, the reason of the access is clear, for instance `BALANCE` needs to access the balance of the target address. Less obvious is the nonce access of several instructions, which is because the EVM uses the nonce (among other things) to check if an account already exists [12, p.4]. For `CALL`, `CALLCODE` and `SELFDESTRUCT`, this is used to calculate the gas costs [12, p.37-38]. For `CREATE` and `CREATE2`, this is used to prevent creating an account at an already active address [12, p.11][4].

In Table 3, we see instructions that can modify the world state.

[4]In the Yellowpaper, the check for the existence of the recipient for `CALL`, `CALLCODE` and `SELFDESTRUCT` is done via the `DEAD` function. For `CREATE` and `CREATE2`, this is done in the condition (113) named `F`.

Instruction	Storage	Balance	Code	Nonce
`SLOAD`	✓			
`BALANCE`		✓		
`SELFBALANCE`		✓		
`CODESIZE`			✓	
`CODECOPY`			✓	
`EXTCODECOPY`			✓	
`EXTCODESIZE`			✓	
`EXTCODEHASH`			✓	
`CALL`		✓	✓	✓
`CALLCODE`		✓	✓	✓
`STATICCALL`			✓	
`DELEGATECALL`			✓	
`CREATE`		✓	✓	✓
`CREATE2`		✓	✓	✓
`SELFDESTRUCT`		✓	✓	✓

Table 2: Instructions that access state. A checkmark indicates, that the execution of this instruction can depend on this state type.

Instruction	Storage	Balance	Code	Nonce
`SSTORE`	✓			
`CALL`		✓		
`CALLCODE`		✓		
`CREATE`		✓	✓	✓
`CREATE2`		✓	✓	✓
`SELFDESTRUCT`	✓	✓	✓	✓

Table 3: Instructions that modify state. A checkmark indicates, that the execution of this instruction can modify this state type.

3.12.2 Causes without code execution

Some state accesses and modifications are inherent to transaction execution. To pay the transaction fees, the sender's balance is accessed and modified. When a transaction transfers some Wei from the sender to the recipient, it also modifies the recipient's balance. To check if the recipient is a contract account, the transaction also needs to access the code of the recipient. Finally, it also verifies the sender's nonce and increments it by one [12, p.9].

3.12.3 Relevant collisions for attacks

The previous sections list possible ways to access and modify the world state. Many previous works have focused on storage and balance collisions, however they did not discuss if or why code and nonce collisions are not important [2], [33]–[36]. Here, we argue, why only storage and balance collisions are relevant for TOD attacks and code and nonce collisions can be neglected.

Following the assumption we made in Section 3.7, in a TOD attack an attacker influences the execution of some transaction T_B, by placing a transaction T_A before it. To have some effect, there must be a write-write or write-read collisions between T_A and T_B. Therefore, our scenario is that we start from some (victim) transaction T_B and try to create impactful collisions with a new transaction T_A.

Let us first focus on the instructions that could modify the codes and nonces that T_B accesses or modifies. As we see in Table 3, these are `SELFDESTRUCT`, `CREATE` and `CREATE2`. Since the EIP-6780 update [37], `SELFDESTRUCT` only destroys a contract if the contract was created in the same transaction. Therefore, `SELFDESTRUCT` can only modify a code and nonce within the same transaction, but cannot be used to attack an already submitted transaction T_B. The instructions to create a new contract, `CREATE` and `CREATE2`, both fail when there is already a contract at the target address [12, p.11]. Therefore, we can only modify the code if the contract previously did not exist. In the case that T_B interacts with some address a that contains no code, the attacker needs `CREATE` or `CREATE2` to create a contract at the address a to force a collision. This is not possible for arbitrary addresses, as the address computation uses the sender's address as an input to a hash function in both cases [12, p.11]. A similar argument can be made about contract creation directly via the transaction and some init code.

Apart from instructions, the nonces of an EOA can also be increased by transactions themselves. T_B could make a `CALL` or `CALLCODE` to the address of an EOA and transfer some Ether. The gas costs for these instructions depend on whether the recipient account already exists or has to be newly created. As such, if T_B makes a `CALL` or `CALLCODE` to a non-existent account, then an attacker could create this account in T_A to reduce the gas costs of the transfer by T_B. We do not consider this an attack, as it only reduces the gas costs for T_B, which likely has no adverse effects.

Therefore, the remaining attack vectors are `SSTORE`, which can modify the storage of an account, and Ether transfers of `CALL`, `CALLCODE`, `SELFDESTRUCT`, which modify the balance of an account.

3.13 Everything is TOD

Our definition of TOD is very broad and marks many transaction pairs as TOD. For instance, if a transaction T_B uses some storage value for a calculation, then the execution likely depends on the transaction that previously has set this storage value. Similarly, when someone wants to transfer Ether, they can only do so when they first received that Ether. Thus, they are dependent on some transaction that gave them this Ether previously.

Proposition 3.13.1: For every transaction T_B after the London upgrade[5], there exists a transaction T_A such that (T_A, T_B) is TOD.

Proof: Consider an arbitrary transaction T_B with the sender being some address $sender$. The sender must pay some upfront cost $v_0 > 0$, because they must pay a base fee [12, p.8-9]. Therefore, we must have $\sigma((\text{'balance'}, sender)) \geq v_0$. This requires that a previous transaction T_A increased the balance of $sender$ to be high enough to pay the upfront cost, i.e. $\text{prestate}\left(\Delta_{T_A}\right)((\text{'balance'}, sender)) < v_0$ and $\text{poststate}\left(\Delta_{T_A}\right)((\text{'balance'}, sender)) \geq v_0$.[6]

When we calculate $\sigma - \Delta_{T_A}$ for our TOD definition, we would set the balance of $sender$ to $\text{prestate}\left(\Delta_{T_A}\right)((\text{balance}, sender)) < v_0$ and then execute T_B based on this state. In this case, T_B would be invalid, as the $sender$ would not have enough Ether to cover the upfront cost. □

Given this property, it is clear that TOD alone is not a useful attack indicator, since every transaction would be considered as having been attacked. In Section 6, we discuss more restrictive definitions.

[5]We reference the London upgrade here, as this introduced the base fee for transactions.

[6]For block validators, their balance could have also increased from staking rewards, rather than a previous transaction. However, this would require that a previous transaction gave them enough Ether for staking in the first place. [38]

4 TOD candidate mining

In this chapter, we discuss how we search for potential TODs in the Ethereum blockchain. We use the RPC from an archive node to obtain transactions and their state accesses and modifications. Then we search for collisions between these transactions to find TOD candidates. Lastly, we filter out TOD candidates that are not relevant to our analysis.

4.1 TOD candidate finding

We make use of the RPC method `debug_traceBlockByNumber`, which allows for replaying all transactions of a block the same way they were originally executed. With the `prestateTracer` config, this method also outputs, which part of the state has been accessed, and using the `diffMode` config, also which part of the state has been modified[7].

By inspecting the source code of the tracers for Reth [39] and the results of the RPC call, we found out that for every touched account, it always includes the account's balance, nonce and code in the prestate. For instance, even when only the balance was accessed, it will also include the nonce in the prestate[8]. Therefore, we do not know precisely which part of the state has been accessed, which can be a source of false positives for collisions.

We store all the accesses and modifications in a database and then query for accesses and writes that have the same state key. As in our definition of collisions, we only match state keys where the first transaction modifies the state. We then use the transactions that cause these collisions as a preliminary set of TOD candidates.

4.2 TOD candidate filtering

Many of the TOD candidates from the previous section are not relevant for our further analysis. To prevent unnecessary computation and distortion of our results, we define which TOD candidates are not relevant and then filter them out.

A summary of the filters is given in Table 4 with detailed explanations in the following sections. The filters are executed in the order as presented in the table and always operate on the output of the previous filter. The only exception is the "Same-value collision"

[7]When running the prestateTracer in diffMode, several fields are only implicit in the response. We need to make these fields explicit for further analysis. Refer to the documentation or the source code for further details.

[8]I opened a pull request to clarify this behavior and now this is also reflected in the documentation[40].

filter, which is directly incorporated into the initial collisions query for performance reasons.

The “Block windows”, “Same senders” and “Recipient Ether transfer” filters have already been used in [3]. The filters “Nonce and code collision” and “Indirect dependency” follow directly from our discussion above. Furthermore, we also applied an iterative approach, where we searched for TOD candidates in a sample block range and manually analyzed whether some of these TOD candidates may be filtered. This approach led us to the “Same-value collisions” and the “Block validators” filter.

Filter name	**Description of filter criteria**
Same-value collision	Drop collision if T_A writes a different value than the value accessed or overwritten by T_B.
Block windows	Drop candidate if T_A and T_B are 25 or more blocks apart.
Block validators	Drop collisions on the block validator’s balance.
Nonce and code collision	Drop nonce and code collisions.
Indirect dependency	Drop candidates (T_A, T_B) with an indirect dependency, e.g. when candidates (T_A, T_X) and (T_X, T_B) exist.
Same senders	Drop candidate if T_A and T_B are from the same sender.
Recipient Ether transfer	Drop candidate if T_B does not execute code.

Table 4: TOD candidate filters sorted by usage order. When a filter describes the removal of collisions, the TOD candidates will be updated accordingly.

4.2.1 Filters

4.2.1.a Same-value collisions

When we have many transactions that modify the same state, e.g. the balance of the same account, they will all have a write-write conflict with each other. The number of TOD candidates grows quadratic with the number of transactions modifying the same state. For instance, if 100 transactions modify the balance of address a, the first transaction has a write-write conflict with all other 99 transactions, the second transaction with the remaining 98 transactions, etc., leading to a total of $\frac{n^2-n}{2} = 4,950$ TOD candidates.

To reduce this growth of TOD candidates, we additionally require for a collision that T_A writes exactly the value that is read or overwritten by T_B. Formally, the following condition must hold to pass this filter:

$$\forall K \in \text{collisions}(T_A, T_B) : \text{poststate}\left(\Delta_{T_A}\right)(K) = \text{prestate}\left(\Delta_{T_B}\right)(K)$$

With the example of 100 transactions modifying the balance of address a, when the first transaction sets the balance to 1234, it only has a write-write conflict with transactions where the balance of a is exactly 1234 before the execution. If all transactions write different balances, this will reduce the amount of TOD candidates to $n - 1 = 99$.

Apart from the performance benefit, this filter also removes many TOD candidates that are potentially indirectly dependent. For instance, let us assume that we removed the TOD candidate (T_A, T_B). By definition of this filter, there must be some key K with $\text{poststate}\left(\Delta_{T_A}\right)(K) \neq \text{prestate}\left(\Delta_{T_B}\right)(K)$, thus some transaction T_X must have modified the state at K between T_A and T_B. Therefore, we also have a collision (and TOD candidate) between T_A and T_X, and between T_X and T_B. This is a potential indirect dependency, which may lead to unexpected results, as argued in Section 3.9.2.

4.2.1.b Block windows

According to a study of 24 million transactions from 2019 [41], the maximum observed time it took for a pending transaction to be included in a block was below 200 seconds. Therefore, when a transaction T_B is submitted, and someone instantly attacks it by creating a new transaction T_A, the inclusion of them in the blockchain differs by at most 200 seconds. We currently add a new block to the blockchain every 12 seconds according to Etherscan [42], thus T_A and T_B are at most $\frac{200}{12} \approx 17$ blocks apart from each other. As the study is already five years old, we use a block window of 25 blocks instead to account for a potential increase in latency since then.

Thus, we filter out all TOD candidates, where T_A is in a block that is 25 or more blocks away from the block of T_B.

4.2.1.c Block validators

In Ethereum, each transaction must pay a transaction fee to the block validator and thus modifies the block validator's balance. This makes each transaction pair in a block a TOD candidate, as they all modify the balance of the block validator's address.

We exclude TOD candidates, where the only collision is the balance of any block validator.

4.2.1.d Nonce and code collisions

We showed in Section 3.12.3 that nonce and code collisions are not relevant for TOD attacks. Therefore, we ignore collisions of this state type.

4.2.1.e Indirect dependency

As argued in Section 3.9.2, indirect dependencies can cause unexpected results in our analysis, therefore we will filter TOD candidates that have an indirect dependency. We will only consider the case, where the indirect dependency is already visible in the normal order and accept that we may miss indirect dependencies. Alternatively, we could also remove a TOD candidate (T_A, T_B) when we there exists a TOD candidate (T_A, T_X) for some intermediary transaction T_X, however this would remove many more TOD candidates.

We already have a model of all direct (potential) dependencies with the TOD candidates. We can build a transaction dependency graph $G = (V, E)$ with V being all transactions and $E = \{(T_A, T_B) \mid (T_A, T_B) \in \text{TOD candidates}\}$. We then filter out all TOD candidates (T_A, T_B) where there exists a path $T_A, T_{X_1}, ..., T_{X_n}, T_B$ with at least one intermediary node T_{X_i}.

Figure 1 shows an example dependency graph, where transaction A influences both X and B and B is influenced by all other transactions. We filter out the candidate (A, B) as there is a path $A \to X \to B$, but keep (X, B) and (C, B).

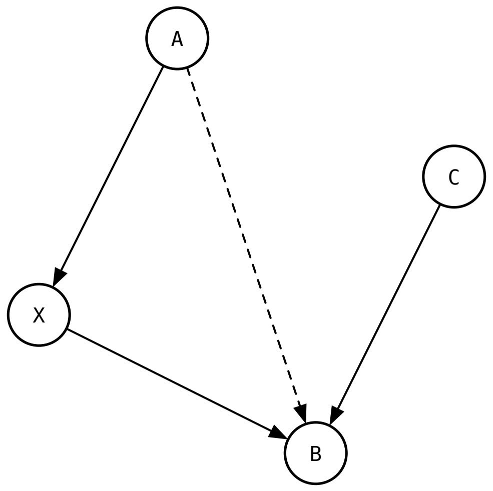

Figure 1: Indirect dependency graph. An arrow from x to y indicates that y depends on x. A dashed arrow indicates an indirect dependency.

4.2.1.f Same sender

If the sender of both transactions is the same, the victim would attack themselves.

To remove these TOD candidates, we use the `eth_getBlockByNumber` RPC method and compare the sender fields for T_A and T_B.

4.2.1.g Recipient Ether transfer

If a transaction sends Ether without executing code, it only depends on the balance of the EOA that signed the transaction. Other entities can only increase the balance of this EOA, which has no adverse effects on the transaction.

Thus, we exclude TOD candidates, where T_B has no code access.

4.3 Experiment

In this section, we discuss the results of applying the TOD candidate mining methodology to a randomly sampled sequence of 100 blocks, different from the block range we used for the filters' development. Refer to Section 9 for the experiment setup.

We mined the blocks from block 19,830,547 up to block 19,830,647, containing a total of 16,799 transactions.

4.3.1 Performance

The mining process took a total of 502 seconds, with 311 seconds used to fetch the data via RPC calls and store it in the database, 6 seconds used to query the collisions in the database, 17 seconds for filtering the TOD candidates and 168 seconds for preparing statistics. If we consider the running time as the total time excluding the statistics preparation, we analyzed an average of 0.30 blocks per second.

We also see that 93% of the running time was spent fetching the data via the RPC calls and storing it locally. This could be parallelized to significantly speed up the process.

4.3.2 Filters

In Table 5, we see the number of TOD candidates before and after each filter, showing how many candidates were filtered at each stage. This shows the importance of filtering as we reduce the number of TOD candidates to analyze from more than 60 million to only 8,127.

Note that this does not directly imply that "Same-value collision" filters out more TOD candidates than "Block windows", as they operate on different sets of TOD candidates. Even if "Block windows" filtered out every TOD candidate, this would be less than "Same-value collision" did, because of the order of filter application.

Filter name	TOD candidates after filtering	Filtered TOD candidates
(unfiltered)	(lower bound) 63,178,557	
Same-value collision	56,663	(lower bound) 63,121,894
Block windows	53,184	3,479
Block validators	39,899	13,285
Nonce collision	23,284	16,615
Code collision	23,265	19
Indirect dependency	16,235	7,030
Same senders	9,940	6,295
Recipient Ether transfer	8,127	1,813

Table 5: This table shows the application of all filters used to reduce the number of TOD candidates. Filters were applied from top to bottom. Each row shows how many TOD candidates remained and were filtered. The unfiltered value is a lower bound, as we only calculated this number afterwards, and the calculation does not include write-write collisions.

4.3.3 Transactions

After applying the filters, 7,864 transactions are part of at least one TOD candidate. This amounts to 46.8% of all transactions marked as potentially TOD with some other transaction. Only 2,381 of these transactions are part of exactly one TOD candidate. At the other end, there exists one transaction that is part of 22 TOD candidates.

4.3.4 Block distance

In Figure 2, we see that most TOD candidates are within the same block. Moreover, the further two transactions are apart, the less likely we are to include them as a TOD candidate. A reason for this may be that having many intermediary transactions makes it more likely to be filtered by our "Indirect dependency" filter. Nonetheless, we can conclude that when using our filters, the block window can be reduced even further without missing many TOD candidates.

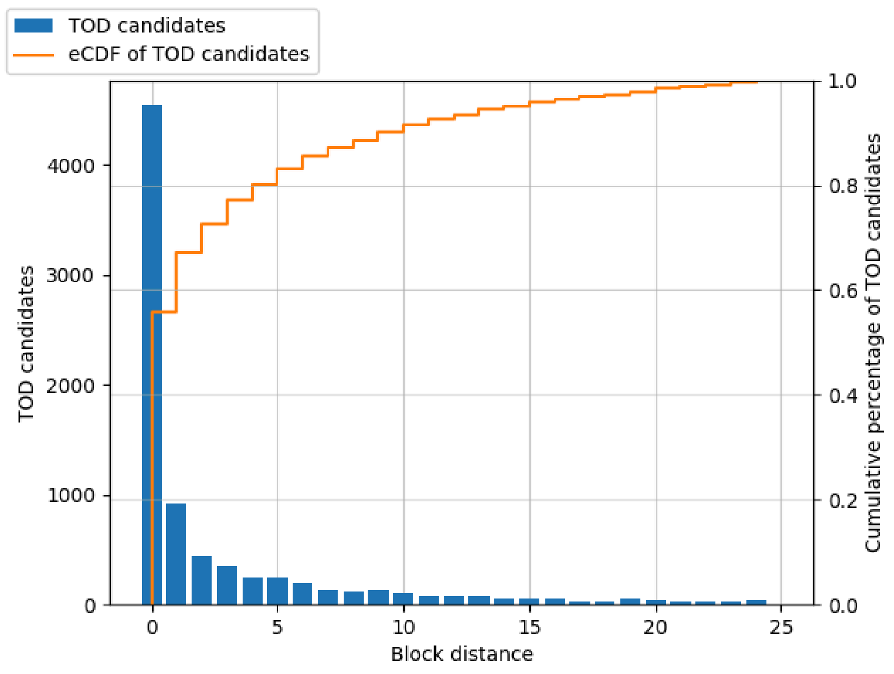

Figure 2: The histogram and the empirical cumulative distribution function (eCDF) of the block distance for TOD candidates. The blue bars show how many TOD candidates have been found, where T_A and T_B are n blocks apart. The orange line shows the percentage of TOD candidates that are at most n blocks apart.

4.3.5 Collisions

After applying our filters, we have 8,818 storage collisions and 5,654 balance collisions remaining. When we analyze how often each account is part of a collision, we see that collisions are concentrated around a small set of accounts. For instance, the five accounts with the most collisions[9] contribute 43.0% of all collisions. In total, the collisions occur in only 1,472 different account states.

Figure 3 depicts how many collisions we get when we only consider the first n collisions for each address. If we set the limit to one collision per address, we end up with 1,472 collisions, which is exactly the number of unique addresses where collisions happened. When we keep 10 collisions per address, we get 3,964 collisions. This criterion already reduces the number of collisions by 73%, while still retaining a sample of 10 collisions for each address.

[9]All of them are token accounts: WETH, DOP, USDT, USDC and CHOPPY

This paper tires to obtain a diverse set of attacks. With such a strong imbalance towards a few contracts, it will take a long time to analyze TOD candidates related to these frequent addresses, and the attacks are likely related and do not cover a wide range of attack types. To prevent this, we define additional deduplication filters in Section 4.4.

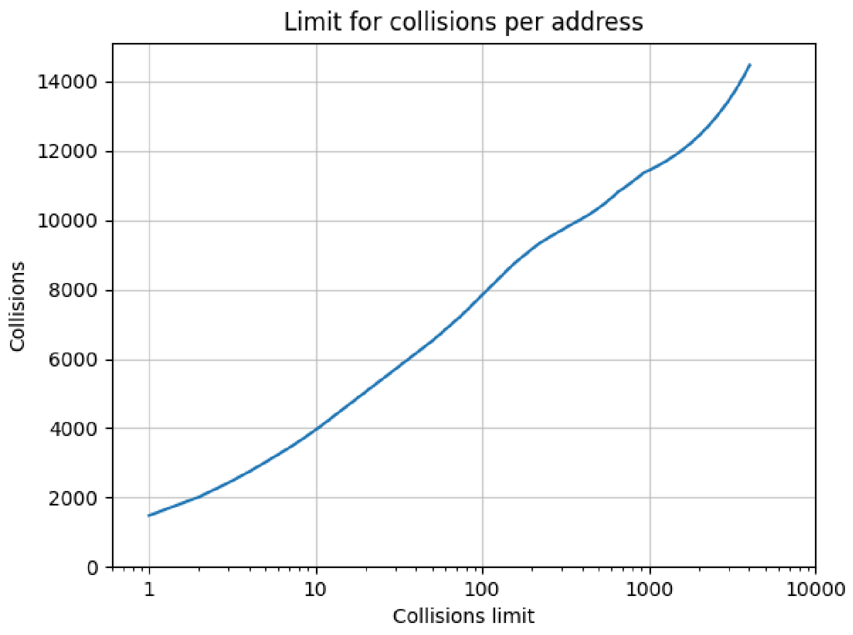

Figure 3: The chart shows how many collisions we have when we limit the number of collisions we include per address. For instance, if we only included 10 collisions for each address we would end up with about 4,000 collisions.

4.4 Deduplication

To reduce the prevalence of specific contracts among the TOD candidates, we randomly pick 10 collisions of each contract and drop the rest. We apply three mechanisms to group similar contracts:

Firstly, we group the collisions by the address where they happened and randomly select 10 collisions from each group. For instance, if many transactions access the balance and code of the same address, we would only retain 10 of these accesses.

Secondly, we also group collisions at different addresses if the addresses share exactly the same code. To do so, we group the collisions by the code hash and sample 10 collisions per code hash.

Finally, instead of matching for exactly the same code, we also group similar codes together. We use the grouping mechanism from [43], where the authors compute a “skeleton” for each code by removing the metadata and the values for `PUSH` instructions. They have shown that codes with the same skeleton mostly yield the same vulnerability detection results. Therefore, we only keep 10 collisions per code skeleton.

4.4.1 Results

We ran the same experiment as in the previous section, but now with the additional deduplication filters. In Table 6, we see that from the initial 8,127 TOD candidates, only 2,320 remain after removing duplicates. Most TOD candidates are already removed by limiting the amount of collisions per address and the other group limits reduce it further.

Filter name	**TOD candidates after filtering**	**Filtered TOD candidates**
(previous filters)	8,127	
Limited collisions per address	2,645	5,482
Limited collisions per code hash	2,435	210
Limited collisions per skeleton	2,320	115

Table 6: This table shows the application of the deduplication filters. We start with the TOD candidates from Table 5 and then apply each deduplication filter.

5 TOD detection

After mining a list of TOD candidates, we now check which of them are actually TOD. We first execute T_A and T_B according to the normal and reverse scenario defined in Section 3.5. Then we compare the state changes of the scenarios to apply the definitions for TOD and approximately TOD.

5.1 Transaction execution via RPC

Let (T_A, T_B) be our TOD candidate. We split the block containing T_B into three sections:

$$\sigma \overset{T_{X_0}}{\to} \dots \overset{T_{X_n}}{\to} \sigma_{X_n} \overset{T_B}{\to} \sigma_{T_B} \overset{T_{Y_0}}{\to} \dots \overset{T_{Y_m}}{\to} \sigma_B$$

In the normal scenario, we want to execute T_B on σ_{X_n} and in the reverse scenario on $\sigma_{X_n} - \Delta_{T_A}$. We use the `debug_traceCall` RPC method for these transaction executions. As parameters, it takes the transaction data, a block number that specifies the block environment and the initial world state, and state overrides that allow us to customize specific parts of the world state. Per default, the method uses the world state *after* executing all transactions in this block, i.e. σ_B. Therefore, we use the state overrides parameter to get from σ_B to σ_{X_n} and $\sigma_{X_n} - \Delta_{T_A}$.

For the normal scenario, we want to execute T_B on σ_{X_n}. Conceptually, we start from σ_B and then undo all transaction changes after T_{X_n} in reverse order, to reach σ_{X_n}. We do this with the state overrides $\sum_{i=m}^{0} \left(-\Delta_{T_{Y_i}}\right) - \Delta_{T_B}$. For the reverse scenario, we also subtract Δ_{T_A} from the state overrides, thus simulating how T_B behaves without the changes from T_A, giving us the state change Δ'_{T_B}.

To execute T_A in the normal scenario we use the same method as for T_B, except that we apply it on the block of T_A. For the reverse scenario, we take the state overrides from the normal scenario and add Δ'_{T_B} to it, simulating how T_A behaves after executing T_B. This yields the state changes Δ'_{T_A}.

5.2 Execution inaccuracy

While manually testing this method, we found that using `debug_traceCall` with state overrides can lead to incorrect gas cost calculations with Erigon[10]. To account for these inaccuracies, we compare the state changes from the normal execution via `debug_traceCall` with the state changes from `debug_traceBlockByNumber`. As we do

[10]See https://github.com/erigontech/erigon/issues/11254.

not provide state overrides to `debug_traceBlockByNumber`, this method should yield the correct state changes, and we can detect differences to our simulation.

If the state changes of a transaction only differ in the balances of the senders and the block validators, we keep TOD candidates containing this transaction. Such differences are to be expected when gas costs vary, as the gas costs affect the priority fee sent from the transaction sender to the block validator. However, if there are other differences, we exclude the transaction from further analysis, as the simulation does not reflect the actual behavior in such cases.

A drawback of this inaccuracy is that we do not detect Ether flows between the senders of T_A and T_B that are TOD. For instance, if the sender of T_A sends one Ether to the sender of T_B in the normal scenario, but two Ether in the reverse scenario, then (T_A, T_B) is TOD. However, our analysis would assume that the Ether changes are due to incorrect gas cost calculations and exclude the TOD candidate from further analysis.

5.3 TOD assessment

We use the state changes Δ_{T_A} and Δ_{T_B} from the normal scenario and Δ'_{T_A} and Δ'_{T_B} from the reverse scenario to check for TOD. For the approximation, we test $\Delta_{T_B} \nsim \Delta'_{T_B}$ and for the exact definition we test $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$.

Algorithm 1 shows, how we perform these state change comparisons. The changed keys, prestates and poststates are obtained from the RPC calls. The black lines show the calculation for the approximation and the blue lines the modifications for the exact definition. For each state key, we compute the change for this key in the normal scenario (d_1), and the change in the reverse scenario (d_2). If the changes differ between the scenarios, we have a TOD.

```
for $K \in \text{changed\_keys}(\Delta_{T_B}) \cup \text{changed\_keys}(\Delta'_{T_B})$
        $\cup\, \text{changed\_keys}(\Delta_{T_A}) \cup \text{changed\_keys}(\Delta'_{T_A})$
    $d_1 = \text{poststate}(\Delta_{T_B})(K) - \text{prestate}(\Delta_{T_B})(K)$
    $d_2 = \text{poststate}(\Delta'_{T_B})(K) - \text{prestate}(\Delta'_{T_B})(K)$
    $d_1 = d_1 + \text{poststate}(\Delta_{T_A})(K) - \text{prestate}(\Delta_{T_A})(K)$
    $d_2 = d_2 + \text{poststate}(\Delta'_{T_A})(K) - \text{prestate}(\Delta'_{T_A})(K)$
    if $d_1 \neq d_2$
        return <TOD>
return <not TOD>
```

Algorithm 1: TOD assessment

5.4 Experiment

We checked all 2,320 TOD candidates we found previously for TOD and approximately TOD. We then compare the results of these, to evaluate how well the approximation performs in practice.

5.4.1 Results

In Table 7, we see the results for both definitions. From the 2,320 TOD candidates we analyzed, slightly more than one third are TOD according to both definitions. For the approximation, 19 TOD candidates cannot be analyzed because of execution inaccuracies. For the exact definition, this number is higher, as we need to execute double the amount of transactions.

With both definitions, for 29% of the TOD candidates, T_B fails because of insufficient funds to cover the transaction fee when it is executed without the state changes by T_A. This can happen when T_A transfers Ether to the sender of T_B, and T_B has less balance than the transaction fee without this transfer. Furthermore, if the execution of T_B consumes more gas without the changes of T_A, it needs to pay a higher transaction fee which can also lead to insufficient funds. In both cases, the existence of T_A enables the execution of T_B, therefore we do not consider these to be TOD attacks and ignore them from further analysis.

Finally, one error occurred when analyzing for the TOD approximation which did not occur with the exact definition. However, this error is not reproducible, potentially being a temporary fault with the RPC requests.

Result	Approximately TOD	TOD
TOD	809	775
not TOD	819	839
inaccurate execution	19	34
insufficient Ether	672	672
error	1	0

Table 7: The results of analyzing TOD candidates for TOD and the approximation of TOD.

5.4.2 Analysis of differences

To understand in which cases the two definitions lead to different results, we manually evaluate the cases where one result was TOD and the other not. To assist the analysis, we let our tool output the relative changes of each transaction in both scenarios. In all the cases, we manually verify that the manual application of Algorithm 1 on the relative changes gives the same result as the automatic application, to ensure the algorithm was correctly implemented.

Our analysis shows that 34 TOD candidates have been marked as approximately TOD but not TOD. As such, we have $\Delta_{T_B} \nsim \Delta'_{T_B}$ and $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \sim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$. In all these cases, the differences of T_A between the normal and reverse scenario balance out the differences of T_B between the normal and reverse scenario. One example is discussed in detail in Appendix B.1.

Further 10 TOD candidates are TOD but not approximately TOD, i.e. $\langle \Delta_{T_A}, \Delta_{T_B} \rangle \nsim \langle \Delta'_{T_A}, \Delta'_{T_B} \rangle$ but $\Delta_{T_B} \sim \Delta'_{T_B}$. In these cases, T_A creates different state changes depending on whether it was executed before or after T_B, thus being TOD. The execution of T_B is not dependent on the transaction order.

A weakness of this comparison is that we use TOD candidates that are tailored for the TOD approximation and therefore TOD candidates that are TOD may be underrepresented. This could be why we found 34 TOD candidates that are approximately TOD but not TOD, while we only found 10 TOD candidates that are TOD but not approximately TOD.

Nonetheless, of the 1,628 TOD candidates labeled as TOD or not TOD according to our approximation, we obtained the same label with the exact TOD definition for 96.4% of these TOD candidates. In the case that TOD transaction pairs are underrepresented in our sample, this still demonstrates that most candidates labeled as approximately TOD are also TOD.

6 TOD attack characteristics

Previously, we noted that the TOD definition is too general to be directly used for attack or vulnerability detection. In this section, we discuss several characteristics of TOD attacks that cover more specific cases than the general TOD definition.

6.1 Attacker gain and victim losses

In Section 3.3, we already discussed how the definition by W. Zhang *et al.* [3] relates to our preliminary definition of TOD. We now present their definition in more detail.

Their definition considers two transaction orderings: $T_A \to T_B \to T_P$ and $T_B \to T_A \to T_P$. When an attack occurs, T_A and T_B are TOD. The transaction T_P is an optional third transaction, which sometimes is required for the attacker to make financial profits. Our study only considers transaction pairs, therefore we adapt their definition and remove T_P from it.

They define an attack to occur when both of the following properties hold:

1. Attacker Gain: "The attacker obtains financial gain in the attack scenario compared with the attack-free scenario."
2. Victim Loss: "The victim suffers from financial loss in the attack scenario compared with the attack-free scenario."

Their attack scenario corresponds to the normal order and the attack-free scenario to the reverse order.

For financial gains and losses, they consider Ether and ERC-20, ERC-721, ERC-777, and ERC-1155 tokens. As an attacker, they consider either the sender of T_A or the contract that T_A calls. The rationale for using the contract that T_A calls is that it may be designed to conduct attacks and temporarily store the profits (see e.g. [4] for more details). The victim is the sender of T_B.

6.1.1 Formalization

The authors of [3] do not provide a precise definition of attacker gain and victim loss, therefore we formalize these definitions. For simplicity, we do not explicitly mention T_A and T_B in all formulas, but assume that we inspect a specific TOD candidate (T_A, T_B) and usages of the normal and reverse scenario refer to these two transactions.

6.1.1.a Assets

We use $\text{Assets}(T_A, T_B)$ to denote a set of assets that occur in T_A and T_B in any of the scenarios. As an asset, we consider Ether and the tokens that implement one of the standards ERC-20, ERC-721, ERC-777 or ERC-1155. Let assets_normal$(C, a) \in \mathbb{Z}$ be the amount of assets C that address a gained or lost by executing both transactions in the normal scenario. Let assets_reverse(C, a) be the counterpart for the reverse scenario.

For example, assume an address a converts 1 Ether to 3,000 USDT tokens in the normal scenario, but in the reverse scenario converts 1 Ether to only 2,500 USDT. The assets that occur are $\text{Assets}(T_A, T_B) = \{\text{Ether}, \text{USDT}\}$. The currency changes are assets_normal$(\text{Ether}, a) = -1$, assets_normal$(\text{USDT}, a) = 3,000$, assets_reverse$(\text{Ether}, a) = -1$ and assets_reverse$(\text{USDT}, a) = 2,500$.

For Ether, we use the `CALL` and `CALLCODE` instructions to compute which addresses gained and lost Ether in a transaction. We do not include the transaction value, as it stays the same regardless of the transaction order[11]. Furthermore, we ignore gas costs because of the inaccuracies described in Section 5.2.

To track the gains and losses for tokens we use the following standardized events:

- ERC-20: `Transfer(address _from, address _to, uint256 _value)`
- ERC-721: `Transfer(address _from, address _to, uint256 _tokenId)`
- ERC-777: `Minted(address operator, address to, uint256 amount, bytes data, bytes operatorData)`
- ERC-777: `Sent(address operator,address from,address to,uint256 amount,bytes data,bytes operatorData)`
- ERC-777: `Burned(address operator, address from, uint256 amount, bytes data, bytes operatorData)`
- ERC-1155: `TransferSingle(address _operator, address _from, address _to, uint256 _id, uint256 _value)`
- ERC-1155: `TransferBatch(address _operator, address _from, address _to, uint256[] _ids, uint256[] _values)`

We only consider calls and event logs if their call context has not been reverted. In Ethereum, a reverted call context means that all changes except for the gas payment are discarded, therefore reverted calls and logs do not influence the gained or lost assets.

6.1.1.b Attacker gain and victim loss

We use the following predicates to express the existence of some asset gain or loss for an address a:

[11] In the course of the evaluation, we actually discover that it would make sense to include the transaction value. See Section 7.2.2.a.

$$\begin{aligned}\text{Gain}(a) &\leftrightarrow \exists C \in \text{Assets}(T_A, T_B) : \text{assets\_normal}(C, a) > \text{assets\_reverse}(C, a)\\ \text{Loss}(a) &\leftrightarrow \exists C \in \text{Assets}(T_A, T_B) : \text{assets\_normal}(C, a) < \text{assets\_reverse}(C, a)\end{aligned}$$

Continuing the previous example of Ether to USDT token conversion, we would have $\text{Gain}(a) = \top$, as a makes more USDT in the normal scenario than in the reverse scenario, and $\text{Loss}(a) = \bot$, as neither for Ether, nor for USDT a has fewer assets in the normal scenario than in the reverse scenario.

However, we also need to consider the case, where both $\text{Gain}(a)$ and $\text{Loss}(a)$ are true. For instance, maybe the attacker gains more USDT tokens but also pays more Ether in the normal scenario. It is not trivial to compare arbitrary assets in Ether, therefore we cannot determine if the lost Ether outweighs the gained tokens. To avoid such cases, we introduce the following two predicates:

$$\begin{aligned}\text{OnlyGain}(a) &\leftrightarrow \text{Gain}(a) \land \neg\ \text{Loss}(a)\\ \text{OnlyLoss}(a) &\leftrightarrow \text{Loss}(a) \land \neg\ \text{Gain}(a)\end{aligned}$$

Note that this only considers assets we explicitly model. In the case that a loses some asset that is not modeled, e.g. a token not implementing any of the above standards, $\text{OnlyGain}(a)$ can be true despite having losses of an unmodeled asset. This is a limitation when not all relevant assets that occur in T_A and T_B are modeled.

With OnlyGain and OnlyLoss we define an attack to occur when the attacker has only advantages in the normal scenario compared to the reverse scenario, and the victim has only disadvantages:

$$\begin{aligned}\text{Attack} \leftrightarrow\ &(\text{OnlyGain}(\text{sender}(T_A)) \lor \text{OnlyGain}(\text{recipient}(T_A)))\\ &\land \text{OnlyLoss}(\text{sender}(T_B))\end{aligned}$$

We note that the definition by [3] is not explicit on how different kinds of assets are compared. As such, our attack formalization may differ from their intention and implementation. This is a best effort to match their implementation and also the definitions of a subsequent [44][12].

6.2 Securify TOD properties

The authors of Securify describe three TOD properties [2]:

- **TOD Transfer**: "[...] the execution of the Ether transfer depends on transaction ordering".

[12]We use the tests in `profit_test.go` [27] and Appendix A of [44] to understand the intended definition.

- **TOD Amount**: “[…] the amount of Ether transferred depends on the transaction ordering”.
- **TOD Receiver**: “[…] the recipient of the Ether transfer might change, depending on the transaction ordering”.

For Ether transfers, they consider only `CALL` instructions. We also use `CALLCODE` instructions, as these can be used to transfer Ether similar to `CALL`s.

The properties can be applied by comparing the execution of a transaction in the normal scenario with the reverse scenario. We say that a property holds for a transaction pair (T_A, T_B) if it holds for at least one of the transactions T_A and T_B, i.e. at least one of the transactions shows attack characteristics.

6.2.1 Formalization

We denote the execution of an instruction as a tuple $(\mathit{Instruction}, \mathit{Loc}, \mathit{Inputs})$. The location Loc is a tuple $(\mathit{ContextAddress}, \mathit{ProgramCounter})$, where $\mathit{ContextAddress}$ is the address that is used for storage and balance accesses when executing the instruction, and $\mathit{ProgramCounter}$ is the byte offset of the instruction in the executed code. Finally, Inputs is a sequence of stack values passed as arguments to the instruction.

Let F_N denote the set of `CALL` and `CALLCODE` instruction executions with a positive value (i.e. $\mathit{Inputs}[2] > 0$) in the normal scenario and F_R the equivalent for the reverse scenario. We exclude calls that have been reverted. For a call execution $C \in F_N$, we denote its location with C_L, the value it transfers with C_v and the recipient of the transfer with C_a.

6.2.1.a TOD Transfer

If there is a location where the number of `CALL`s differ between the normal and the reverse scenario, we say that TOD Transfer is fulfilled:

$$\text{TOD-Transfer} \leftrightarrow \exists \mathit{loc} : |\{C \in F_N \mid C_L = \mathit{loc}\}| \neq |\{C \in F_R \mid C_L = \mathit{loc}\}|$$

6.2.1.b TOD Amount

If there is a location and a value where the number of `CALL`s differ between the normal and the reverse scenario, we say that TOD Amount is fulfilled:

$$\begin{aligned}\text{TOD-Amount} \leftrightarrow{}& \neg\text{TOD-Transfer}\\ &\wedge \exists \mathit{loc}, v : |\{C \in F_N \mid C_L = \mathit{loc} \wedge C_v = v\}| \neq |\{C \in F_R \mid C_L = \mathit{loc} \wedge C_v = v\}|\end{aligned}$$

We exclude cases where TOD Transfer is fulfilled, as TOD Amount would always be fulfilled if TOD Transfer is fulfilled.

6.2.1.c TOD Receiver

We define TOD Receiver analogously to TOD Amount, except that we use the `address` input instead of the `value`:

$$\begin{aligned}\text{TOD-Receiver} \leftrightarrow{} & \neg\text{TOD-Transfer} \\ & \land \exists loc, a : |\{C \in F_N \mid C_L = loc \land C_a = a\}| \neq |\{C \in F_R \mid C_L = loc \land C_a = a\}|\end{aligned}$$

6.3 ERC-20 multiple withdrawal

Finally, we also consider ERC-20 multiple withdrawal attacks, which we already discussed in Section 3.2.2. The ERC-20 standard defines that the following events must be emitted when an approval takes place and when tokens are transferred [25].

- `Approval(address _owner, address _spender, uint256 _value)`
- `Transfer(address _from, address _to, uint256 _value)`

As a pattern to detect ERC-20 multiple withdrawal attacks we require the following conditions to be true:
1. Executing T_A in the normal scenario must emit an event `Transfer`(v, a, x) at address t.
2. Executing T_B in the normal scenario must emit an event `Approval`(v, a, y) at address t.
3. Executing T_B in the reverse scenario must emit an event `Approval`(v, a, y) at address t.

The variable a represents the attacker address, v the victim address x the transferred value and y the approved value. We require that the events are not reverted.

As shown in Table 1, executions of `transferFrom` and `approve` can be TOD because `approve` overwrites the currently approved value with the newly approved value. While this behavior is standardized in [25], other methods may prevent ERC-20 multiple withdrawal attacks by making a relative increase of the approved value rather than overwriting it. To ensure that there is indeed an overwrite, we require that the approval in the normal scenario is equal to the one in the reverse scenario. If there were a relative change of the approval, the approved value y would differ.

6.4 Trace analysis

To check for the TOD characteristics, we use the same approach to compute state overrides for the normal and reverse scenario as in Section 5.1. The `debug_traceCall` method allows the definition of a custom tracer in Javascript that can process each execution step. We use this tracer to track `CALL` and `CALLCODE` instructions and token events.

The Javascript tracer is described in Appendix C. When executing a transaction, it returns all non-reverted `CALL`, `CALLCODE`, `LOG0`, `LOG1`, `LOG2`, `LOG3` and `LOG4` instructions and their inputs. We parse the call instructions to obtain Ether changes and the log instructions for token changes and ERC-20 `Approval` events. The results are used to check for the previously defined characteristics.

7 Evaluation

In this section, we evaluate the methods proposed above. We use a dataset from [3] as a ground truth to evaluate our TOD candidate mining, the TOD detection and the detection of the attacker gain and victim loss characteristic. For the Securify and ERC-20 multiple withdrawal characteristics, we rely solely on a manual evaluation.

The ground truth dataset contains 6,765 attacks in the block range of 11,299,000 to 11,300,000. From these attacks, 5,601 contain no profit transaction, which we excluded from our definition of the attacker gain and victim loss property. The study by [4] also investigated this block range, and the attacks they found are a subset of the 6,765 attacks [3]. Therefore, showing that our method works well for this ground truth indirectly also shows that it works well for the results of [4].

First, we combine the TOD candidate mining, the TOD detection, and the TOD attack analysis method to analyze this block range. The results are discussed in Section 7.2, where we evaluate our method for false positives. Afterwards, we compare the results of each step individually with the ground truth to check for false negatives.

7.1 Evaluation limitations

We note that in our evaluation, we verify the correctness of the normal scenario, however our verification of the reverse scenario is limited as we do not have access to a ground truth for comparison.

For the normal scenario, we can directly compare it with data from the blockchain, as the executions in the normal scenario should equal the executions that happened on the blockchain. We will use Etherscan [26] to access this blockchain data.

Contrary, for the reverse scenario, we simulate a transaction order that did not occur on the blockchain. We can verify that our normal and reverse scenarios are suitable for detecting TOD attacks by comparing our results to the ground truth dataset. However, we only have the results given in the dataset and not the exact executions of the reverse scenario. Therefore, when we encounter differences we cannot conduct an in-depth analysis to understand why differences occur between our method and the ground truth. To allow future research making in-depth comparisons we provide traces for all cases that we manually analyze (see Section 9.2), which contain each execution step for the normal and reverse scenarios.

We also compare our normal scenario with the reverse scenario and evaluate where these executions differ. We do so in Section 7.6.1.b, where we verify that the first difference between the normal and reverse scenario matches the state calculations we perform according to the definitions of the normal and normal scenario.

7.2 Overall evaluation

We mined TOD candidates in the 1,000 blocks starting at 11,299,000, which resulted in 14,500 TOD candidates. From those, the TOD detection reported 2,959 as TOD. For 280 of these transaction pairs, we found an attacker gain and victim loss.

We compare the TOD candidates, TODs, and TOD attacks we found against the ground truth in Table 8. Our mining procedure marks 115 of the attacks in the ground truth as TOD candidates. From the 115 TOD candidates, 95 are detected as TOD, and of those, 85 are marked as an attack.

When mining the TOD candidates we drop 98% of the ground truth attacks. The following steps drop another 26% of the attacks. We evaluate the reasons for this in the following sections, where we evaluate each component individually.

This section focuses on the 195 attacks we found that are not part of the ground truth.

In ground truth	**TOD candidate**	**TOD**	**Attacker gain and victim loss**
Yes	115	95	85
No	14,385	2,864	195

Table 8: Comparison of results with the 5,601 attacks from the ground truth. The first row shows how many of the 5,601 attacks in the ground truth are also found by our analysis at the individual stages. The second row shows the results our method found, which are not in the ground truth.

7.2.1 Block window filter

W. Zhang *et al.* [3] only consider transactions within block windows of size three. If transactions are three more blocks apart from each other, they are not part of their analysis. We use a block window of size 25, therefore finding more attacks.

Of the 195 attacks we find that are not in the ground truth, only 19 are within a block window of size 3.

7.2.2 Manual analysis of attacks

We manually evaluate the 19 attacks to check if the attacker gain and victim loss property holds. We perform the following steps for each attack:

1. We manually parse the execution traces of the normal and reverse scenario for calls and events related to the attacker and victim accounts.
2. We compute the attacker gain and victim loss property based on these calls and events.
3. For the normal scenario, we verify that the calls and logs for the attacker and victim accounts are equal to those that occurred on the blockchain.

In all 19 cases, the manual evaluation shows that the attacker gain and victim loss property holds and that the relevant calls and logs in the normal scenario match those on the blockchain. However, we notice two shortcomings in our definition of the attacker gain and victim loss property.

7.2.2.a Definition shortcomings

Firstly, we argued that the transaction value is independent of the transaction order, because it is part of the transaction itself. However, when a transaction is reverted, the value is not sent to the receiver. Therefore, the transfer of the transaction value may depend on the transaction order. If we considered the transaction value in the calculation, six of the 19 attacks would be false positives.

Secondly, in five cases, we have a loss for the sender of T_A (the attacker's EOA), while we have only gains for the recipient of T_A (considered the attacker's bot in this case). Our definition considers the attacker gain fulfilled for the attacker's bot and ignores the loss of the attacker's EOA. If we considered them together, we may have different results in such cases.

7.3 Evaluation of Securify and ERC-20 multiple withdrawal characteristics

In the overall analysis, we also analyze the 2,959 transaction pairs that are TOD for the Securify and ERC-20 multiple withdrawal characteristics.

We find that 626 transaction pairs fulfill the TOD Transfer characteristic, 244 TOD Amount, and 1 TOD Receiver. Moreover, we have 15 that fulfill our definition of ERC-20 multiple withdrawal. As the ground truth does not cover these characteristics, we manually analyze samples of each.

7.3.1 Manual evaluation of TOD Transfer

We take a sample of 20 transaction pairs that fulfill TOD Transfer. Our tool outputs the locations at which there is a different amount of calls in the normal and reverse scenario. For each sample, we verify the first call location it shows for T_A and T_B. To do so, we manually check the execution traces of the normal and reverse scenario for this location and extract the relevant calls. We further verify that these calls match the calls in the normal scenario are equal to those on the blockchain.

We find that in all cases, the TOD transfer property holds for T_B, and only in one case it holds additionally for T_A.

In 9 of the cases, T_B makes a `CALL` in the normal scenario that is reverted in the reverse scenario. As our definition only considers calls that are not reverted, these fulfill TOD Transfer.

In 8 further cases, T_B makes a `CALL` in the normal scenario but makes no `CALL` at this location in the reverse scenario. In the 3 remaining cases, T_B makes a `CALL` in the reverse scenario but makes no `CALL` in the normal scenario at this location.

We also observe that the locations are often the same. For instance, in five of the cases, the location we analyze is the address `0x7a250d5630b4cf539739df2c5dacb4c659f2488d` at program counter `15784`. When inspecting all 626 transaction pairs that fulfill TOD Transfer we find this location 86 times. Considering that we limit similar collisions to a maximum of 10, this implies that different collisions affect the same functionality.

7.3.2 Manual evaluation of TOD Amount

We take a sample of 20 transaction pairs that fulfill TOD Amount. Similar to the TOD Transfer evaluation, we manually verify the first location reported by our tool. For TOD Amount, we verify that in both scenarios there exists a call at this location, but with different values.

The evaluation shows that the property holds in all cases for T_B, and in 3 cases also for T_A. In 12 cases, the amount of Ether sent is increased in the reverse scenario, and in 11 cases, it is decreased.

Again, we observe many calls happening at the same location. Of the 20 call locations we analyze, the location is 16 times at the address `0x7a250d5630b4cf539739df2c5dacb4c659f2488d` at program counter `15784`.

7.3.3 Manual evaluation of TOD Receiver

We evaluate the one transaction pair we found for TOD Receiver similar to how we evaluate TOD Amount, except that we now verify whether the receiver of the Ether transfer changed. Our evaluation shows that this is indeed the case. By inspecting the traces, we can see that in the normal scenario the receiver address is loaded from a different storage slot than in the normal scenario, resulting in different recipients of the Ether transfer.

7.3.4 Manual evaluation of ERC-20 multiple withdrawal

We evaluate all 15 transaction pairs where our tool reports an ERC-20 multiple withdrawal attack. Our tool outputs pairs of `Transfer` and `Approval` events that should fulfill the definition. For each case, we manually evaluate the first of these pairs by verifying that the `Transfer` event exists in T_A in the normal scenario and the `Approval` event exists in T_B in the normal and reverse scenario. We further verify that the logs in the normal scenario are equal to those on the blockchain.

While we confirm that all of them fulfill the definition we provide for the ERC-20 multiple withdrawal attack, none of them actually is an attack.

7.3.4.a Definition shortcomings

Firstly, our definition does not require that the `Transfer` and `Approval` events have positive values. In nine cases we find an `Approval` event that approves 0 tokens and in one case we find a transfer of 0 tokens. These should be excluded from the definition.

Moreover, in 14 cases T_A contains an `Approval` event for the tokens that are transferred in T_A. As such, T_A does not use any previously approved tokens, but approves the token itself.

7.4 Evaluation of TOD candidate mining

In this section, we analyze why 98% of the attacks in the ground truth are not reported as TOD candidates, and whether the TOD candidate filters work as intended.

We rerun the TOD candidate mining and count the number of attacks from the ground truth that are in the TOD candidates before and after each filter is applied. Therefore, we know how many of the attacks were removed by which filter.

In Table 9, we see that most filters do not filter out any attack from the ground truth. However, they still filter out 500,141 other TOD candidates, thus significantly reducing the search space for further analysis without affecting the attacks we can find.

Furthermore, Table 9 shows that only one attack is filtered because there is no collision between the accessed and modified states of T_A and T_B. This TOD candidate is filtered, because the second transaction of the filtered TOD candidate is part of block 11,300,000, which is not part of the blocks we analyze[13].

The filters “Same-value collision” and “Indirect dependency” remove 4,275 TOD candidates with potential indirect dependencies. Finally, our deduplication filters remove an-

[13]In [3], the dataset is described as originating from an analysis of 1,000 blocks. Block 11,300,000 would be the $1,001$-th block, thus we assume an off-by-one error.

other 1,210 TOD candidates. In the following subsections, we evaluate whether these filters fulfill their intention.

Filter name	**TOD candidates after filtering**	**Filtered TOD candidates**	**Ground truth attacks after filtering**	**Filtered ground truth attacks**
(unfiltered)			5,601	
Collision	(unknown)		5,600	1
Same-value collision	638,313	(unknown)	3,537	2,063
Block windows	422,384	215,929	3,537	0
Block validators	288,264	134,120	3,537	0
Nonce collision	220,687	67,577	3,537	0
Code collision	220,679	8	3,537	0
Indirect dependency	161,062	59,617	1,325	2,212
Same senders	100,690	60,372	1,325	0
Recipient Ether transfer	78,555	22,135	1,325	0
Limited collisions per address	17,300	61,255	199	1,126
Limited collisions per code hash	14,996	2,304	123	76
Limited collisions per skeleton	14,500	496	115	8

Table 9: Comparison of all filtered TOD candidates with filtered attacks from the ground truth. Each row shows how many TOD candidate and attacks are filtered by this filter. TOD candidates before filtering for same-value collisions were not compute because of performance limitations.

7.4.1 Evaluation of indirect dependency filters

The "Same-value collision" and "Indirect dependency" filters both target TOD candidates with indirect dependencies, as these may lead to unexpected analysis results (see Section 3.9.2).

We evaluate, for how many of the removed attack TOD candidates (T_A, T_B), there exists an intermediary transaction T_X, such that both (T_A, T_X) and (T_X, T_B) are TOD. In such cases, any reordering that moves T_A after T_X or T_X after T_B may influence how T_A and T_B execute. While our filters also remove indirect dependencies which require more than one intermediary transaction (e.g. $T_A \to T_{X_1} \to T_{X_2} \to T_B$), we limit our evaluation to only one intermediary transaction for performance reasons.

We rerun the TOD candidate mining until the "Indirect dependency" filter would be executed. For 1,720 of the 4,275 TOD candidates (T_A, T_B) we evaluate, we find another two TOD candidates (T_A, T_X) and (T_X, T_B). These TOD candidates show a potential indirect dependency of (T_A, T_B) with the one intermediary transaction T_X. We do not evaluate the remaining 2,555 TOD candidates, which either have an indirect dependency with multiple intermediary transactions, or have an indirect dependency where one of the TOD candidates (T_A, T_X) or (T_X, T_B) has already been filtered.

We run or TOD detection on the 1,720 (T_A, T_X) TOD candidates and the 1,720 (T_X, T_B) TOD candidates. We find that in 1,319 cases both (T_A, T_X) and (T_X, T_B) being TOD. In 159 cases, at least one analysis failed. In the remaining 242 cases, at least one of the TOD candidates (T_A, T_X) or (T_X, T_B) is confirmed not to be TOD.

In summary, we show that in at least 1,319 of the 4,275 cases where we filtered out a an attack of the ground truth, there exists a transaction that is TOD with both T_A and T_B of this attack and thus potentially interferes with the TOD simulation.

7.4.2 Evaluation of duplicate limits

The filters "Limited collisions per address", "Limited collisions per code hash" and "Limited collisions per skeleton" aim to reduce the amount of TOD candidates without reducing the diversity of the attacks we find.

For our evaluation, we do not directly measure the diversity of the attacks. Instead, we evaluate how well the attacks that were not filtered cover the attacks that were filtered. To measure the coverage, we use collisions. We say, that a TOD candidate (T_A, T_B) is covered by a set of TOD candidates $\left\{\left(T_{C_0}, T_{D_0}\right), \ldots, \left(T_{C_n}, T_{D_n}\right)\right\}$ if the following condition holds:

$$\text{collisions}(T_A, T_B) \subseteq \bigcup_{0 \le i \le n} \text{collisions}\left(T_{C_i}, T_{D_i}\right)$$

For this analysis, we only consider collisions that remain after applying all previous filters.

From the 1,210 attacks that were removed by duplicate limits, we have 703 that are covered by the remaining attacks. Thus, if we combine the collisions of the 115 remaining attacks, we have the same collisions as if we included these 703 covered attacks. From the 703 covered attacks, we can match at least[14] 504 removed attacks (T_A, T_B) with a remaining attack (T_C, T_D), such that $\text{collisions}(T_A, T_B) \subseteq \text{collisions}(T_C, T_D)$. Thus, in 504 cases a removed attack is covered by exactly one remaining attack.

We also notice that attacks are concentrated around the same collisions. When we take the top three attacks that were not removed by duplicate limits, we already cover 373 of the 1,210 attacks we removed.

Our calculations of coverages are lower limits, as we only consider the 115 remaining attacks from the ground truth but not the 195 attacks we find that are not in the ground truth. All attacks we find are subject to the duplicate limit, therefore some ground truth attacks may have been removed while keeping an attack not in the ground truth that has similar collisions.

7.5 Evaluation of TOD detection

To evaluate our TOD detection method, we run it on the attacks from the ground truth.

From the 5,601 attacks, our method finds that 4,827 are TOD and 4,857 approximately TOD. We do not manually compare the differences of the TOD detection with the approximation, as we already did so in Section 5. However, that for the attacks from the ground truth one can use the TOD approximation without loosing attacks.

There are 774 attacks that our method misses. For 20 of those an error occurred while analyzing for TOD[15] and for 296 we detected execution inaccuracies (see Section 5.2) and stopped the analysis.

From the remaining 458 attacks, we find that most have the metadata "out of gas" in the ground truth dataset. Attacks with this "out of gas" label account for 97.6% of the attacks we do not find, while they only account for 19.1% of the 5,601 attacks in the ground truth.

[14]We use a naive algorithm to detect collision coverage, which does not minimize the required attacks for coverage. Thus, the number of attacks covered by a single other attack is a lower bound.

[15]18 of the errors are caused by a bug in Erigon, where it reports negative balances for accounts for some transactions (fixed in v2.60.3). 2 of them were caused by connection errors.

7.5.1 Manual evaluation of attacks labeled “out of gas”

According to the dataset description, this label refers to a *gas estimation griefing attack*, which is described in [3]. The authors consider such an attack to occur when T_B runs out of gas in the normal scenario but not in the reverse scenario.

We manually inspect a sample of 20 attacks and find that in 12 attacks, T_B is indeed reverted according to Etherscan. In these cases, our simulation method further shows that T_B is reverted in both scenarios. As T_B also revertes in the reverse scenario, these cases are no gas estimation griefing attacks according to our simulation.

In the remaining 8 cases, our method reports no reverts in either scenario. For one case, Etherscan reports that T_B had an internal out of gas error, which was caught without reverting the whole transaction. Therefore, at least the 7 cases where T_B did not revert in the normal scenario are no gas estimation griefing attacks.

As such, it is unclear if this label classifies these attacks as gas estimation griefing attacks. Furthermore, it appears that attacks with this label do not necessarily fulfill the attacker gain and victim loss property. The dataset usually describes the profits of an attacker and losses of a victim in each attack. However, 347 of the 1,043 attacks with the “out of gas” label do not contain a description of the victim losses. However, this description exists for all attacks without the “out of gas” label. In summary, it is unclear how we should interpret these attacks from the ground truth and thus we ignore them from further analysis.

7.5.2 Manual evaluation of attacks not labeled “out of gas”

We manually check the remaining 11 attacks that our method does not report as TOD.

We check if these are caused by bugs in the RPC method implementation by rerunning the analysis with a Reth archive node, in addition to the Erigon archive node we use for our experiments. In two cases, using Reth we report them as TOD because of the same balance changes as reported in the ground truth, showing the inaccuracies from Section 5.2.

Furthermore, we compare the traces of the instruction executions between the scenarios. For 8 attacks, the traces in the normal scenario are equal to those in the reverse scenario, therefeore no write-read or read-write TOD has occurred. By inspecting the state changes in Etherscan, we also rule out write-write TODs, where both transactions write to the same storage slot. As such, these are indeed TOD accocrding to the traces of our simulation.

Finally, for one attack T_B reverts in both scenarios. The ground truth dataset reports token changes in the reverse scenario, therefore our execution must differ from theirs, which we do not further investigate.

7.6 Evaluation of TOD attack analysis

We run our TOD attack analysis on the 5,601 attacks from the ground truth. Our analysis reports an attacker gain and victim loss in 4,524 of the cases. In 19 cases we encountered the same errors as for the TOD checking. In another 152 cases, we detect execution inaccuracies. In the remaining 907 cases, our analysis runs without failures but reports different results than in the ground truth.

From these 907 cases, 850 are labeled as "out of gas" in the ground truth. As discussed in Section 7.5.1, it seems that these do not necessarily fulfill the attacker gain and victim loss property. Therefore, we do not investigate these cases. We manually evaluate 10 of the 56 cases without the "out of gas" label.

7.6.1 Manual evaluation of attacks

7.6.1.a Evaluation of profit calculations

We verify that the transaction pair does not fulfill the attacker gain and victim loss property according to the execution traces of our normal and reverse scenarios. In each case, we disprove the property by manually parsing the calls and logs and calculating the profits and losses.

In five cases, there is a victim gain according to our traces. In three cases, we calculate an attacker loss. In the two other cases, the traces of the attacker's transaction behave identical in the normal and reverse scenarios. We show that one of these cases is not TOD in Appendix B.2.

7.6.1.b Evaluation of reverse scenario

We further want to verify that our tool correctly executes the reverse scenario.

For each case, we pick one of the transactions. For this transaction, we compare the n-th executed instruction of the normal scenario with the n-th executed instruction of the reverse scenario. We start with $n = 0$ and continue until we find a difference between the executions. For the comparison, we use the current EVM stack, memory, program counter, gas, and call context depth.

In one case, we do not find any difference, as the transactions are not TOD. In the other nine cases, the first difference is after executing the `SLOAD` instruction, which loads a value from the storage.

When we analyze the execution of the attacker's transaction (T_A), we execute it on the states σ and $\sigma + \Delta'_{T_B}$. Because we observer that `SLOAD` returns different values for σ and $\sigma + \Delta'_{T_B}$, the accessed storage slot should be modified by Δ'_{T_B}. We verify that in the normal scenario, the result of the `SLOAD` is equal to the value this storage slot had before executing T_A according to Etherscan. For the reverse scenario, we compare it against the last `SSTORE` of the execution of T_B in the reverse scenario, i.e. the value that Δ'_{T_B} changes it to.

We approach the verification of the accessed storage values of T_B similarly. For T_B we use the states σ_{X_n} for the normal scenario and $\sigma_{X_n} - \Delta_{T_A}$ for the reverse scenario. We compare the result of the `SLOAD` in the normal scenario with the value it had before executing T_B according to Etherscan. For the reverse scenario, we compare it with the value it had before executing T_A according to Etherscan. We notice that in all these cases, T_A wrote a value to this storage slot that is different from the one that T_B reads in the normal scenario. Therefore, there must be intermediary transactions that changed this storage value between T_A and T_B, possibly causing an indirect dependency.

In summary, we verify that at least the first difference between the normal and reverse scenarios is in accordance with the definition of the normal and reverse scenarios.

7.6.1.c Evaluation of indirect dependencies

As our previous evaluation shows, there are several attacks where an intermediary transaction modifies a storage slot that is written by T_A and accessed by T_B, potentially creating an indirect dependency. In this section, we additionally evaluate for a specific kind of indirect dependency.

When we simulate an attack (T_A, T_B), we execute T_B in the reverse scenario on the state $\sigma_{X_n} - \Delta_{T_A}$. Therefore, for all state keys $K \in \text{changed\_keys}\left(\Delta_{T_A}\right)$ we use the value at $\text{prestate}\left(\Delta_{T_A}\right)(K)$ and for the other state keys $K \notin \text{changed\_keys}\left(\Delta_{T_A}\right)$ we use $\sigma_{X_n}(K)$.

We now consider an intermediary transaction T_X with the state changes Δ_{T_X}, where $\text{changed\_keys}\left(\Delta_{T_X}\right)$ contains some keys that are changed by T_A and also other keys that T_A does not change. When we execute T_B on $\sigma_{X_n} - \Delta_{T_A}$, we only overwrite some of the changes of T_X and keep the other changes. Therefore, T_B executes on a state where state changes of T_X are only partially included.

Our evaluation shows that in 2 of the 10 cases, we can indeed find an intermediary transaction T_X, such that at T_B accesses at least one change of T_X that is overwritten by computing $\sigma_{X_n} - \Delta_{T_A}$, and one change of T_X that is not overwritten. Thus, in these cases, T_B uses a possibly incoherent state. We further verify that (T_A, T_X) and (T_X, T_B)

are both TOD. However, we do not investigate, how the partial revert of T_X influences the transaction execution.

7.6.1.d Unmodeled token events

In one case, the attacker's transaction emits a `Deposit` event. This event is not part of the token standards we model in our definition, therefore our profit calculations ignore this event.

This `Deposit` event is emitted by the WETH token when someone converts Ether to WETH tokens. Our analysis only assesses a loss of Ether, but not a gain of WETH tokens. If we modeled the `Deposit` event, we would also mark this transaction pair as an attack.

By inspecting the source code at [27], we find that they also detect `Deposit` and `Withdrawal` events when they are emitted by the address of the WETH token.

7.7 Performance evaluation

The evaluation of the 1,000 blocks took a total of 75 minutes, averaging to 4.5 seconds per block.

For the TOD candidate mining, we spent 41 minutes fetching the state changes of the 1,000 blocks and inserting them into a database, another 13 minutes filtering the TOD candidates, and 3 minutes for other tasks.

For the TOD detection and TOD attack analysis, we fetch the state changes and transactions and store them in memory for faster access. Because the state changes are already in our RPC cache, these two steps combined only took 5 minutes.

After fetching the state changes and transactions, we ran the TOD detection and TOD attack analysis using 16 threads, enabling us to make multiple RPC requests in parallel. To check the 14,500 TOD candidates for TOD it took 11 minutes, an average of 44 milliseconds per TOD candidate. The attack analysis of the 2,959 TOD transaction pairs took 4 minutes, averaging to 77 milliseconds per analysis.

Compared with [3], our analysis took 4.5 seconds per block, while they report an average of 7.5 seconds per block. However, we cannot directly compare this, as their hardware specifications differ from our setup and in our case the transaction execution is outsourced to an archive node of which we do not know the hardware specifications. Moreover, [3] only reports an average for their whole analysis, and it is not clear if e.g. the vulnerability localization performed in this work is included in this time measurement.

8 Discussion

This thesis proposes a method to simulate transaction order dependencies. We precisely define this simulation process and discuss its advantages and disadvantages. Our evaluation shows that it can be used to detect TOD and several attack characteristics, finding more than 80% of the attacks from a previous work.

Nonetheless, we note that our simulation method and the methods of two related studies have drawbacks that can lead to analysis results that do not match the execution that happened on the blockchain or are distorted by the influence of intermediary transactions. The method by [4] removes intermediary transactions for the simulation. On the downside, this may create results that differ from the blockchain even in the normal transaction order. On the upside, the different orderings can then be compared without the potential influence of intermediary transactions. The methods by [3] and us produce results that are equal to the blockchain in the normal order, but can suffer influences from intermediary transactions in the reverse order.

We discuss when influences from intermediary transactions can occur with our method, and thus, we are able to avoid such cases. However, future work may continue to reduce the influence of intermediary transactions on TOD simulations or analyze the tradeoffs between existing methods.

9 Data availability and reproducibility

9.1 Tool

The program used to run the experiments is available at https://github.com/TOD-theses/t_race. It can be run with Python or Docker and requires an archive node that supports the debug namespace, including JS tracing for the attack analysis. Refer to the documentation on the repository for more details on using it.

The TOD candidates deduplication relies on randomness. To allow reproducibility, the program sets a seed for the randomness before executing the randomized deduplication step.

9.2 Data availability

The experiment results and evaluation artifacts produced by this thesis are available at https://github.com/TOD-theses/t_race_results. This includes the outputs of the tool executions, post-processed data and the evaluation samples with corresponding notes.

9.3 Experiment setup

The experiments were performed on Ubuntu 22.04.04, using an AMD Ryzen 5 5500U CPU with 6 cores and 2 threads per core and an SN530 NVMe SSD. We used a 16 GB RAM with an additional 32 GB swap file.

For the RPC requests, we did not use our own archive node, but relied on a free service by [45], which uses Erigon 2.59.3 [46] according to the `web3_clientVersion` RPC method. In the evaluation, we also refer to the usage of a Reth instance for a few TOD checks. We use a public Reth 1.0.4 instance for this [47]. We used a local cache to prevent repeating slow RPC requests [48]. The cache was initially empty for experiments that measure the running time.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] R. Rahimian, S. Eskandari, and J. Clark, "Resolving the Multiple Withdrawal Attack on ERC20 Tokens," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2019, pp. 320–329. doi: 10.1109/EuroSPW.2019.00042.

[2] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. Vechev, "Securify: Practical Security Analysis of Smart Contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2018, pp. 67–82. doi: 10.1145/3243734.3243780.

[3] W. Zhang *et al.*, "Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation," *IEEE Transactions on Software Engineering*, vol. 49, pp. 3630–3646, 2023, doi: 10.1109/TSE.2023.3270117.

[4] C. F. Torres, R. Camino, and R. State, "Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021, pp. 1343–1359. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/torres

[5] P. Daian *et al.*, "Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 910–927. doi: 10.1109/SP40000.2020.00040.

[6] Y. Wang, P. Zuest, Y. Yao, Z. Lu, and R. Wattenhofer, "Impact and User Perception of Sandwich Attacks in the DeFi Ecosystem," in *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, ACM, 2022, pp. 1–15. doi: 10.1145/3491102.3517585.

[7] D. Perez and B. Livshits, "Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited," in *30th USENIX Security Symposium (USENIX Security 21)*, USENIX Association, 2021, pp. 1325–1341. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/perez

[8] C. Ferreira Torres, A. K. Iannillo, A. Gervais, and R. State, "The Eye of Horus: Spotting and Analyzing Attacks on Ethereum Smart Contracts," in *Financial Cryptography and Data Security*, N. Borisov and C. Diaz, Eds., Springer Berlin Heidelberg, 2021, pp. 33–52. doi: 10.1007/978-3-662-64322-8_2.

[9] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, “{TXSPECTOR}: Uncovering Attacks in Ethereum from Transactions,” in *29th USENIX Security Symposium (USENIX Security 20)*, USENIX Association, 2020, pp. 2775–2792. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya

[10] S. Wu *et al.*, “Time-travel Investigation: Toward Building a Scalable Attack Detection Framework on Ethereum,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 3, pp. 1–33, 2022, doi: 10.1145/3505263.

[11] T. Chen *et al.*, “SODA: A Generic Online Detection Framework for Smart Contracts,” in *Proceedings 2020 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2020. doi: 10.14722/ndss.2020.24449.

[12] G. Wood, “Ethereum: A secure decentralised generalised transaction ledger. Paris version.” Accessed: Jul. 10, 2024. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf

[13] S. Tikhomirov, “Ethereum: state of knowledge and research perspectives,” in *Foundations and Practice of Security: 10th International Symposium (FPS 2017)*, A. Imine, J. M. Fernandez, J.-Y. Marion, L. Logrippo, and J. Garcia-Alfaro, Eds., in Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 206–221. doi: 10.1007/978-3-319-75650-9_14.

[14] “History and Forks of Ethereum.” Accessed: Jul. 10, 2024. [Online]. Available: https://ethereum.org/en/history/

[15] smlXL, “EVM Codes.” Accessed: Jul. 10, 2024. [Online]. Available: https://www.evm.codes/

[16] “Gasper.” Accessed: Jul. 11, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/gasper/

[17] S. Eskandari, S. Moosavi, and J. Clark, “SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain,” in *Financial Cryptography and Data Security*, A. Bracciali, J. Clark, F. Pintore, P. B. Rønne, and M. Sala, Eds., Cham: Springer International Publishing, 2020, pp. 170–189. doi: 10.1007/978-3-030-43725-1_13.

[18] L. Heimbach, L. Kiffer, C. Ferreira Torres, and R. Wattenhofer, “Ethereum's Proposer-Builder Separation: Promises and Realities,” in *Proceedings of the 2023 ACM on Internet Measurement Conference*, in IMC '23. ACM, 2023, pp. 406–420. doi: 10.1145/3618257.3624824.

[19] “Nodes and clients.” Accessed: Jul. 11, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/nodes-and-clients/

[20] "Ethereum JSON-RPC Specification." Accessed: Jul. 11, 2024. [Online]. Available: https://ethereum.github.io/execution-apis/api-documentation/

[21] "go-ethereum: debug Namespace." Accessed: Jul. 11, 2024. [Online]. Available: https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug

[22] "RPC daemon." Accessed: Jul. 11, 2024. [Online]. Available: https://erigon.gitbook.io/erigon/advanced-usage/rpc-daemon

[23] "Reth Book." Accessed: Jul. 11, 2024. [Online]. Available: https://reth.rs/jsonrpc/debug.html

[24] T. Chen *et al.*, "TokenScope: Automatically Detecting Inconsistent Behaviors of Cryptocurrency Tokens in Ethereum," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2019, pp. 1503–1520. doi: 10.1145/3319535.3345664.

[25] F. Vogelsteller and V. Buterin, "ERC-20: Token Standard." Accessed: Aug. 12, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-20

[26] Etherscan, "Ethereum (ETH) Blockchain Explorer." Accessed: Aug. 21, 2024. [Online]. Available: https://etherscan.io/

[27] W. Zhang *et al.*, "erebus-redgiant." Accessed: Jul. 13, 2024. [Online]. Available: https://github.com/Troublor/erebus-redgiant/tree/4544163f0c6a369b35c3237851f482d240fa7bbd

[28] C. F. Torres, R. Camino, and R. State, "Frontrunner Jones." Accessed: Jul. 13, 2024. [Online]. Available: https://github.com/christoftorres/Frontrunner-Jones/tree/84e98dae4ab37fad7629433fe3ad41967152431f

[29] "Erigon DB Walkthrough." Accessed: Aug. 12, 2024. [Online]. Available: https://github.com/erigontech/erigon/blob/9b19cd542008d4de3eb267df3c606b2203284ed6/docs/programmers_guide/db_walkthrough.MD

[30] "Reth Database." Accessed: Aug. 12, 2024. [Online]. Available: https://github.com/paradigmxyz/reth/blob/269ba896369c6fcea10e046a124a1992a56af300/docs/design/database.md

[31] P. Rebuffo, "Erigon 3 (Alpha 1), the first all-in-one EVM-node on the efficient software frontier, is live.." Accessed: Aug. 12, 2024. [Online]. Available: https://erigon.tech/erigon-3-alpha-1-the-first-all-in-one-evm-node-on-the-efficient-software-frontier-is-live/

[32] C. Ma, W. Song, and J. Huang, "TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts," in *Proceedings of the 31st ACM*

Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, in ESEC/FSE 2023. ACM, 2023, pp. 947–959. doi: 10.1145/3611643.3616281.

[33] X. Wang, J. Sun, C. Hu, P. Yu, B. Zhang, and D. Hou, "EtherFuzz: Mutation Fuzzing Smart Contracts for TOD Vulnerability Detection," *Wireless Communications and Mobile Computing*, vol. 2022, p. e1565007, 2022, doi: 10.1155/2022/1565007.

[34] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena, "Exploiting the laws of order in smart contracts," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2019. ACM, 2019, pp. 363–373. doi: 10.1145/3293882.3330560.

[35] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ACM, 2016, pp. 254–269. doi: 10.1145/2976749.2978309.

[36] S. Munir and C. Reichenbach, "TODLER: A Transaction Ordering Dependency anaLyzER - for Ethereum Smart Contracts," in *2023 IEEE/ACM 6th International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, 2023, pp. 9–16. doi: 10.1109/WETSEB59161.2023.00007.

[37] G. Ballet, V. Buterin, and D. Feist, "EIP-6780: SELFDESTRUCT only in same transaction." Accessed: Jul. 14, 2024. [Online]. Available: https://eips.ethereum.org/EIPS/eip-6780

[38] "Proof-of-stake rewards and penalties." Accessed: Jul. 31, 2024. [Online]. Available: https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/rewards-and-penalties/

[39] Paradigm, "revm-inspectors." Accessed: Jul. 14, 2024. [Online]. Available: https://github.com/paradigmxyz/revm-inspectors/tree/b9850ffe4d67aadc46cba5e3798bee459a01a560

[40] "go-ethereum: Built-in tracers." Accessed: Jul. 14, 2024. [Online]. Available: https://geth.ethereum.org/docs/developers/evm-tracing/built-in-tracers#prestate-tracer

[41] L. Zhang, B. Lee, Y. Ye, and Y. Qiao, "Evaluation of Ethereum End-to-end Transaction Latency," in *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, 2021, pp. 1–5. doi: 10.1109/NTMS49979.2021.9432676.

[42] Etherscan, "Ethereum Average Block Time Chart." Accessed: Jul. 14, 2024. [Online]. Available: https://etherscan.io/chart/blocktime

[43] M. di Angelo and G. Salzer, “Bytecode Skeletons for Sample Selection in the Analysis of Blockchain Programs,” in *Proceedings of the 6th IEEE International Conference on Blockchain and Cryptocurrency*, 2024.

[44] W. Zhang *et al.*, “Nyx: Detecting Exploitable Front-Running Vulnerabilities in Smart Contracts,” in *2024 IEEE Symposium on Security and Privacy (SP)*, IEEE Computer Society, 2024, p. 149–150. doi: 10.1109/SP54263.2024.00146.

[45] Nodies, “Web3 RPC Platform.” Accessed: Aug. 20, 2024. [Online]. Available: https://www.nodies.app/

[46] “Erigon Documentation.” Accessed: Jul. 16, 2024. [Online]. Available: https://erigon.gitbook.io/erigon/

[47] Paradigm, “Reth.” Accessed: Aug. 20, 2024. [Online]. Available: https://reth.paradigm.xyz/

[48] Fuzzland, “ETH RPC Cache Layer.” Accessed: Jul. 16, 2024. [Online]. Available: https://github.com/fuzzland/cached-eth-rpc

A. Overview of Generative AI Tools Used

I used Grammarly to improve the readability of the text. The whole work was analyzed and I applied several small suggestions.

B. Case studies

B.1. Analysis of definition differences

Here, we present one example for Section 5.4.2 that is approximately TOD but not TOD.

For the following two transactions:
- T_A: `0xa723f53edcae821203572a773b8f1b5cf5c008a734794ee2acae771540363f11`
- T_B: `0x5aa39f4ff79f6653fdb0165a92fcb55e024ae8d5b8dba67c0b6e4c153ea4a8d4`

Both transactions changed a specific storage slot. Our tool outputs the following changes:
- T_B (normal): `+0`
- T_A (normal): `+0x1c7400000000000000000000000000000000000000000000000000000000`
- T_B (reverse): `+0x1c7400000000000000000000000000000000000000000000000000000000`
- T_A (reverse): `+0`

We see, that in both scenarios, the value increases by `0x1c7400000000000000000000000000000000000000000000000000000000`, therefore considering both transactions it is not TOD. However, if we only consider T_B, we would observe a TOD, as T_B changes the storage slot differently in the scenarios (`+0` vs `+0x1c7400000000000000000000000000000000000000000000000000000000`).

In our manual analysis of all cases, this information is enough to say that the application of our definitions was correct, assuming that the state changes outputted by the tool are correct. To further understand, why such changes occur in practice, we analyzed this transaction pair in more detail.

Using Etherscan, we see that both transactions emit a `UsdPerTokenUpdated` event with the parameters `value: 0x429d069189e0000` and `timestamp: 0x663c689f`. Furthermore, it shows for the storage slot at transaction T_A:

- **Before**: `0x663c4c2b00000000000000000000000000000000000000000429d069189e0000`
- **After**: `0x663c689f00000000000000000000000000000000000000000429d069189e0000`

We observe, that the value after T_A is composed of the timestamp and the value of the emitted event. As both transactions emitted the same event with this value and timestamp, it is likely, that both transactions set the value of this storage slot to `0x663c689f00000000000000000000000000000000000000000429d069189e0000`. For T_A, this led to a state change of this storage slot. As T_B is executed after T_A, the storage slot was already at the target value and no change is recorded for T_B. In the reverse scenario, T_B is executed first and therefore we observe a state change here. And similarly for T_A we now record no state change.

The code that updates the storage slot is shown below, located at address `0x8c9b2efb7c64c394119270bfece7f54763b958ad`. In line 5 we see the assignment to the storage slot and in line 9 the logged event. Both transactions have the same values for `update.usdPerToken` and `block.timestamp`, therefore the value assigned to `s_usdPerToken[update.sourceToken]` is the same in both cases.

```
1 contract PriceRegistry {
2   // ...
3   function updatePrices(/* ... */) {
4     // ...
5     s_usdPerToken[update.sourceToken] = Internal.TimestampedPackedUint224({
6       value: update.usdPerToken,
7       timestamp: uint32(block.timestamp)
8     });
9     emit UsdPerTokenUpdated(update.sourceToken, update.usdPerToken, block.timestamp);
10   }
11 }
```

B.2. Analysis of TOD

We analyze one of the cases where a TOD candidate (T_A, T_B) is not TOD according to our definition, but is reported as an attack by [3]. The transaction T_A is 0x5cf84067556e7db37fd0279ec3bfe227d71758786cb53f1cc24e20f8afd9f8d8 and T_B is 0xd24cffe4cd2dd7c89cc7ec3d38f44f4563d184b5fa9a952b46358a8a8e8176cc.

To evaluate if this TOD candidate is TOD, we start by determining the collisions of T_A and T_B. If the execution of T_A influences T_B or vice versa, it must be at a state key that one of them modifies and the other accesses or modifies as well. For this case study, we evaluate $\left(W_{T_A} \cap R_{T_B}\right) \cup \left(W_{T_A} \cap W_{T_B}\right) \cup \left(R_{T_A} \cap W_{T_B}\right)$ to obtain the collisions rather than collisions(T_A, T_B), as we do not want to rely on the TOD approximation underlying the definition of collisions(T_A, T_B).

We use the `debug_traceTransaction` method to obtain the accessed state keys R_{T_A} and R_{T_B} (refer to the Section 9.2 for the data). We then compare these state modifications W_{T_A} and W_{T_B} shown on Etherscan and only find a collision at the balance of the WETH contract. Therefore, the only way T_A may influence T_B, and vice versa, is by modifying and accessing the balance of the WETH contract.

In Table 2, we see the instructions that can access balances. The instructions `CALL`, `CALLCODE`, `CREATE`, `CREATE2` and `SELFDESTRUCT` access a balance by sending Ether from the caller to the recipient. These instructions can behave differently depending on whether enough Ether is available. This is not the case, because the transactions transfer less than 3 Ether from the WETH account to another account, but the WETH account had more than 6 million Ether at this time according to the `eth_getBalance` RPC method.

The remaining two instructions that can cause TOD when accessing balances are `BALANCE` and `SELFBALANCE`. We manually inspect the execution traces in the normal scenario to see if these instructions are used to access the balance of the WETH contract.

According to our normal scenario traces, T_A has three executions of `SELFBALANCE` and T_B has two executions of `SELFBALANCE`, however none of these are lookup the balance of the WETH contract. The traces also show no occurrence of the `BALANCE` instruction.

We further verify that our normal scenario traces execute the same instructions as on Etherscan. We compare the number of execution steps from our traces, which matches with those shown by Etherscan (15171 and 9164), thus we assume we execute the same instructions as on the blockchain.

In summary, we ruled out that any instruction accesses the balance of the WETH contract. As this is the only state key that one transaction writes and the other reads or writes, T_A and T_B, these transactions cannot be TOD.

C. Javascript tracer

We use the following javascript tracer to extract `CALL` instructions and emitted token events. The `step` function is executed for each instruction. In case a `CALL` or `CALLCODE` instruction is found we append data to `this.calls` and for `LOG0`, `LOG1`, `LOG2`, `LOG3` or `LOG4` instruction is found we append it to `this.logs`.

To detect reverted calls, we check in the `exit` function if an error occurred. As an error reverts the current call context and all of its children, we store a mapping of each call context to its children in `children_of`. When reverting a call context, we can then recursively mark all child contexts as reverted.

The `result` function is called when the tracing has finished. We first check if the overall transaction is reverted. Then we return the calls and logs for which their call context has not been reverted.

```
{
  calls: [],
  logs: [],
  call_context_stack: [0],
  call_context_counter: 0,
  reverted_call_contexts: [],
  children_of: {},
  location: function(log) {
    return {
        'address': toHex(log.contract.getAddress()),
        'pc': log.getPC(),
    }
  },
  enter: function(callFrame) {
    current_call_context =
this.call_context_stack[this.call_context_stack.length - 1]
    this.call_context_counter += 1
    this.call_context_stack.push(this.call_context_counter)
    if (!this.children_of[current_call_context]) {
      this.children_of[current_call_context] = []
    }

this.children_of[current_call_context].push(this.call_context_counter)
  },
  exit: function(frameResult) {
    context_id = this.call_context_stack.pop(this.call_context_counter)
    error = frameResult.getError()
    if (error) {
      this._revert(context_id)
    }
  },
  _revert: function(id) {
    // revert context and all of its sub contexts
    this.reverted_call_contexts.push(id)
    children = this.children_of[id] || []
    for (child_id of children) {
      this._revert(child_id)
    }
  },
  step: function(log, db) {
    opcode = log.op.toNumber()
    if (opcode == 0xF1 || opcode == 0xF2) {
        this.calls.push({
            'op': opcode,
            'sender': toHex(log.contract.getAddress()),
            'to': toHex(toAddress(log.stack.peek(1).toString(16))),
            'value': log.stack.peek(2).toString(16),
            'location': this.location(log),
            'call_context_id':
this.call_context_stack[this.call_context_stack.length - 1],
        })
    }
    else if (opcode >= 0xA0 && opcode <= 0xA4) {
        offset = log.stack.peek(0).valueOf()
        size = log.stack.peek(1).valueOf()
        data = toHex(log.memory.slice(offset, offset + size))
        topics_amount = opcode - 0xA0
        topics = []
        for (i = 0; i < topics_amount; i++) {
            topics.push(log.stack.peek(2 + i).toString(16).padStart(64,
"0"))
        }
        this.logs.push({
            'topics': topics,
            'data': data,
            'address': toHex(log.contract.getAddress()),
            'location': this.location(log),
            'call_context_id':
this.call_context_stack[this.call_context_stack.length - 1],
        })
    }
  },
  fault: function(log, db) {},
  result: function(ctx, db) {
    if (ctx.error) {
      this._revert(0)
    }
    logs = this.logs.filter(log => !
this.reverted_call_contexts.includes(log['call_context_id']))
    calls = this.calls.filter(call => !
this.reverted_call_contexts.includes(call['call_context_id']))
    return {
      "gas": ctx.gasUsed,
      "calls": calls,
      "logs": logs,
      "reverted_call_contexts": this.reverted_call_contexts,
    };
  }
}
```