Informatics

T-RACE: Eine Analyse von race condition Angriffen bei Ethereum Transaktionen

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: Ass.Prof.in Dipl.-Ing.in Mag.a rer.soc.oec. Dr.in techn. Monika di Angelo
Mitwirkung: Ao.Univ.Prof. Dr. Gernot Salzer

Wien, 1. Jänner 2001

Othmar Lechner	Monika di Angelo

TU WIEN Informatics

T-RACE: Tracing race condition attacks between Ethereum transactions.

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: Ass.Prof.in Dipl.-Ing.in Mag.a rer.soc.oec. Dr.in techn. Monika di Angelo
Assistance: Ao.Univ.Prof. Dr. Gernot Salzer

Vienna, January 1, 2001

Othmar Lechner	Monika di Angelo

Erklärung zur Verfassung der Arbeit

Othmar Lechner

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

Wien, 1. Jänner 2001

Othmar Lechner

Danksagung

Ihr Text hier.

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

Enter your text here.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Contributions 1

2 Background **3**
2.1 Ethereum 3
2.2 World State 3
2.3 EVM 4
2.4 Transactions 4
2.5 Blocks 5
2.6 Transaction submission 5
2.7 Transaction execution 6
2.8 Nodes 7
2.9 RPC 7

3 Transaction order dependency **9**
3.1 Approaching TOD 9
3.2 Motivating examples 9
3.3 Relation to previous works 10
3.4 Imprecise definitions 10
3.5 TOD definition 13
3.6 State collisions 16
3.7 TOD candidates 16
3.8 Causes of state collisions 17
3.9 Everything is TOD 19

4 TOD candidate mining **21**
4.1 TOD candidate finding 21
4.2 TOD candidate filtering 22

4.3 Experiment . . . 25
4.4 Deduplication . . . 27

5 Trace analysis **29**

6 TOD Attack results **31**

7 Tool benchmarking **33**
7.1 Systematic Literature Review . . . 33
7.2 Result . . . 33

8 Data availability **35**

9 Reproducibility **37**
9.1 Tool . . . 37
9.2 Randomness . . . 37
9.3 Experiment setup . . . 37

Overview of Generative AI Tools Used **39**

List of Figures **41**

List of Tables **43**

List of Algorithms **45**

Bibliography **47**

CHAPTER 1

Introduction

Enter your text here.

TBD.

1.1 Contributions

- Precise definition of TOD in the context of blockchain transaction analysis.
- Theoretical discussion of TOD, including compilation of instructions that can cause TOD.
- Methodology to mine potential TOD transaction pairs using only the RPC interface of an archive node, rather than requiring local access to it.

CHAPTER 2

Background

This chapter gives background knowledge on Ethereum, that is helpful to follow 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". [Woo24, 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*. [Tik18]

2.2 World State

Similar to [Woo24, 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 following data [Woo24, 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, a smaller unit of Ether.
- *storage*: The storage allows contract accounts to persistently store information across transactions. It is a key-value mapping where both, key and value, are 256 bytes long. For EOAs, this is empty.

[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 nonce and code.

- *code*: For contract accounts, the code is a sequence of EVM instructions.

We denote the world state as σ, the account state of an address a as $\sigma(a)$ and the nonce, balance, storage and code as $\sigma(a)_n$, $\sigma(a)_b$, $\sigma(a)_s$ and $\sigma(a)_c$ respectively. For the value at a storage slot k we write $\sigma(a)_s[k]$. We will also an alternative notation $\sigma(K)$, where we combine the identifiers of a state value to a single key K, which simplifies further definitions. We have the following equalities between the two notations:

$$\begin{aligned}
\sigma(a)_n &= \sigma((\text{"}nonce\text{"}, a)) \\
\sigma(a)_b &= \sigma((\text{"}balance\text{"}, a)) \\
\sigma(a)_c &= \sigma((\text{"}code\text{"}, a)) \\
\sigma(a)_s[k] &= \sigma((\text{"}storage\text{"}, a, k))
\end{aligned}$$

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 gas. When it runs out of gas, the execution will halt. [Woo24, p.14] For instance, this prevents execution of infinite loops, as it would use infinitely much gas and thus exceed the gas limit.

Most EVM instructions are formally defined in. [Woo24, p.30-38] However, the Yellowpaper currently does not include the changes from the Cancun upgrade [noa24d], therefore we will also refer to the informal descriptions available on evm.codes. [sml24]

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 following data relevant to our work:

- *sender*: The address of the transaction sender[2].
- *recipient*: The destination address.
- *value*: The value of Wei that should be transferred from the sender to the recipient.

[2]The sender is implicitly given through a valid signature and the transaction hash. [Woo24, 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.

- *gasLimit*: The maximum number 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. [Woo24, p.4-5]

For every transaction the sender must pay a *transaction fee*. This is composed of a *base fee* and a *priority fee*. Every transaction must pay the base fee. The amount of Wei will be reduced 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. [Woo24, p.8]

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

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. [noa23] 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. [Woo24, p.8]

Each block consists of a block header and a sequence of transactions. We denote the nth block of the blockchain as B_n and the sequence of transactions it includes as $T(B_n) = (T_1, T_2, \ldots, T_m)$.

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 frontrunning, as it gives nodes time to react on a transaction before it becomes part of the blockchain. [EMC20]

A different approach, the Proposer-Builder Separation (PBS) has become 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. [HKFTW23]

2.7 Transaction execution

In Ethereum, transaction execution is deterministic. [Woo24, 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 $\sigma \xrightarrow{T} \sigma\prime$, 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 $poststate(\Delta\sigma_T) = \sigma\prime$ the world state after the execution of T.

For two state changes Δ_{T_A} and Δ_{T_B}, we say that $\Delta_{T_A} = \Delta_{T_B}$ if they changed the same set of state fields and the pre- and poststate for these changed fields is equal, otherwise $\Delta_{T_A} \neq \Delta_{T_B}$. For instance, if both Δ_{T_A} and Δ_{T_B} modified only the storage slot $\sigma(a)_s[k]$, and both changed it from the value x to the value y, we would call them equal. If Δ_{T_B} changed it from $x\prime$ to y, or from x to $y\prime$ or even modified a different storage slot $\sigma(a)_s[k\prime]$, we would say $\Delta_{T_A} \neq \Delta_{T_B}$.

We define the set of changed state keys as:

$$changed\_keys(\Delta) := \{K | prestate(\Delta)(K) \neq poststate(\Delta)(K)\}$$

We let the equality $\Delta_{T_A} = \Delta_{T_B}$ be true if following holds, else $\Delta_{T_A} \neq \Delta_{T_B}$:

$$\begin{aligned} changed\_keys(\Delta_{T_A}) &= changed\_keys(\Delta_{T_B}) \\ \forall K \in changed\_keys\colon prestate(\Delta_{T_A})(K) &= prestate(\Delta_{T_B})(K) \\ \text{and } poststate(\Delta_{T_B})(K) &= poststate(\Delta_{T_B})(K) \end{aligned}$$

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 $poststate(\Delta_T)$. 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 $prestate(\Delta_T)$. Formally, these definitions are as follows:

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

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

For instance, if transaction T changed the storage slot 1234 at address 0xabcd from 0 to 100, then $(\sigma + \Delta_T)(\text{0xabcd})_s[1234] = 100$ and $(\sigma - \Delta_T)(\text{0xabcd})_s[1234] = 0$. For all other storage slots we have $(\sigma + \Delta_T)(a)_s[k] = \sigma(a)_s[k] = (\sigma - \Delta_T)(a)_s[k]$.

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 to reproduce the state and transactions at any block. [noa24e]

2.9 RPC

Execution clients implement the Ethereum JSON-RPC specification. [noa24a] 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 [noa24c][noa24h][noa24g].

CHAPTER 3

Transaction order dependency

In this chapter we discuss our definition of transaction order dependency (TOD) and various properties that come with it. We first lay out the idea of TOD with a basic definition and then show several shortcomings of this simple definition. Based on these insights, we construct a more precise definition that we will use for our analysis.

3.1 Approaching TOD

Intuitively, a pair of transactions (T_A, T_B) is transaction order dependent (TOD), if the original execution order leads to a different result than a reordered execution order. In formal terms, we write this as 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 executing 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 *reversed* execution order.

3.2 Motivating examples

TBD.

Add a motivating example for write-read TOD (e.g. TOD-recipient) and for write-write TOD (e.g. ERC-20 approval).

3.3 Relation to previous works

In [TCS21] the authors do not provide a formal definition of TOD. However, 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 intuitive 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 instruction, instead of the resulting state. This would miss attacks where the same instructions were executed, but the operands for these instructions in the second transaction changed because of the first transaction.

In [ZWC+23a], they 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 the 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 $T_v \to T_a \to T_a^p$. They monitor the transactions to check if the execution order impacts financial gains, which we will discuss later in more detail.

Add a reference when the section exists

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, 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 intuitive definition of TOD, and the related definitions shown above, are not precise on the semantics of a reordering of transactions and their executions. These make it impossible to apply exactly the same methodology without analyzing the source code related to the papers. We detect three issues, where the definition is not precise enough and show how these were differently interpreted by the two papers.

For the analysis of the tools by [ZWC+23a] and [TCS21], we will use the current version of the source codes, [ZWC+23b] and [TCS22] respectively.

3.4.1 Intermediary transactions

To analyze the TOD (T_A, T_B), we are interested in how T_A affected T_B. Our intuitive definition did not specify how to handle transactions that occurred between T_A and T_B, which we will name *intermediary transactions*.

For instance, let us assume that there was one transaction T_X in 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 clearly could depend on both, T_A and T_X. When we are interested in the impact of T_A on T_B, we need to define what happens with T_X.

For executing the normal order, we would 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 to σ_{AXB}).

When executing the reverse order, we could make following choices:

1. $\sigma \xrightarrow{T_B} \sigma_B \xrightarrow{T_A} \sigma_{BA}$, which ignores T_X and thus may impact 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.

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

In [ZWC+23a], this impact of the intermediary transactions is acknowledged and caused 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, [TCS21], does not mention this issue at all.

Should I move the technical aspects to an appendix? e.g. only discussing the results here, but moving the code analysis to the appendix?

Code analysis of [ZWC+23a]

As shown in their algorithm 1, they take as input all the executed transactions. They use these transactions and their results in the `searchVictimGivenAttack` method, where `ar` represents the attack transaction and result and `vr` represents the victim transaction and result.

For the normal execution order ($T_a \to T_v$), they simply 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 ($T_v \to T_a$), 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` built in `StartSession`, which seems to be a graph where each transaction points to the previous transaction from the same EOA. From this graph, it takes 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 [ZWC+23a] executes all intermediary transactions for the normal order. However, for the reverse order, they only execute intermediary transactions that are also sent by the victim, but do not execute any other intermediary transactions.

Code analysis of [TCS21]

In the file `displacement.py`, they replay the normal execution order at the lines 154-155, and the reverse execution order at the lines 158-159. 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 transactoins (T_A, T_B), it can be, that these are not part of the same block. The execution of these transactoins can 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 different block environment than on the blockchain may alter their behaviour. From our intuitive TOD definition, it is not clear which block environment(s) we use when replaying the transactions in normal and reverse order.

Code analysis of [ZWC+23a]

The block environment used to execute all transactions is contained in `ar.VmContext` and as such corresponds to the block environment of T_a. This means 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 different block environment.

Code analysis of [TCS21]

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 different block environment than 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 is.

Code analysis of [ZWC$^+$23a]

The initial state used to execute the first transaction is `ar.State`, which corresponds to the state directly before executing T_a. This includes all previous transactions of the same block.

Code analysis of [TCS21]

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 impacted by a previous transaction in the same block.

3.5 TOD definition

To address the issues above, we will provide a more precise definition for TOD.

Definition 1 (TOD)**.** *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.

We say, that (T_A, T_B) is TOD if and only if executing $(\sigma_{X_n} - \Delta_{T_A}) \xrightarrow{T_B} \sigma_{B'}$ produces a state change $\Delta_{T_{B'}}$ with $\Delta_{T_B} \neq \Delta_{T_{B'}}$.

Intuitively, we take the world state exactly before T_B was executed, namely σ_{X_n}. We then record the state changes Δ_{T_B} from executing T_B directly on σ_{X_n}, the same way it was executed on the blockchain. Then we simulate what would have happened if T_A was not executed before T_B by removing its state changes and executing T_B on $\sigma_{X_n} - \Delta_{T_A}$. If we observe different state changes for T_B when executed with and without the changes of T_A, then we know that T_A has an impact on T_B and conclude TOD between T_A and T_B. If there are no differences between Δ_{T_B} and $\Delta_{T_{B'}}$, then T_B behaves the same regardless of T_A and there is no TOD.

We chose to compare the two executions on the state changes $\Delta_{T_B} \neq \Delta_{T_{B'}}$, rather than on the resulting states $\sigma_B \neq \sigma_{B'}$, to detect a wider range of TODs. Comparing on $\sigma_B \neq \sigma_{B'}$ would be sufficient to detect *write-read* TODs, where the first transaction writes some state and the second transaction accesses this state and outputs a different result because of this. However, we are also interested into *write-write* TODs, where T_A writes some state and T_B overwrites the same state with a different value, thus hiding the change by T_A.

For example, let T_A write the value *aaaa* to some storage, s.t. we have $\sigma_{X_n}(a)_s[k] = aaaa$, and T_B write *bbbb* to the same storage, s.t. we have $\sigma_B(a)_s[k] = bbbb$. When executing T_B last, the world state would have *bbbb* at this storage slot, and when executing T_A last, it would be *aaaa*. Therefore, the resulting world state is dependent on the order of T_A and T_B. To check for this case, we compare the prestates of each change in Δ_{T_B} and $\Delta_{T_{B'}}$. In our example, when executing T_B on σ_{X_n} we would have $prestate(\Delta_{T_B})(a)_s[k] = aaaa$ (as the changes from T_A are included in this scenario), but when executing on $\sigma_{X_n} - \Delta_{T_A}$ we have $prestate(\Delta_{T_{B'}})(a)_s[k] = 0000$ (as the changes from T_A are undone in this scenario). Therefore, checking for inequality between the prestates from the state changes Δ_{T_B} and $\Delta_{T_{B'}}$ can detect write-write TODs.

Our definition does not include *read-write* TODs, i.e. we do not check whether executing T_B before T_A would have an impact on T_A. We focus on detecting TOD attacks, in which the attacker tries to insert a transaction prior to some transaction T and impact the behaviour of T with this. Therefore, we assume that the first transaction tries to impact the second transaction, and not ignore the other way round.

3.5.1 Definition strengths

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 $\sigma_{X_n} - \Delta_{T_A} = \sigma + \Delta_{T_A} + (\sum_{i=0}^{i=n} \Delta_{T_{X_i}}) - \Delta_{T_A}$ and then execute T_B on this state. Using state changes allows us to check if T_A and T_B are TOD with only one transaction execution, despite including the effects of arbitrary many intermediary transactions.

If we want to check n transactions for TOD, we could execute all n transactions to obtain their state changes. There are $\frac{n^2-n}{2}$ transaction pairs, thus if we wanted to test each pair for TOD we would end up with a total of $n + \frac{n^2-n}{2} = \frac{n^2+n}{2}$ transaction executions. Similar to [TCS21] and [ZWC$^+$23a], we can filter irrelevant transactions pairs to drastically reduce the search space.

Similarity to blockchain executions

With our definition, the state change Δ_{T_B} from the normal execution is equivalent to the state change that happend on the blockchain. Also, the reversed order is closely related to the state from the blockchain, as we start with σ_{X_n} and only modify the relevant parts for our analysis. Furthermore, we prevent effects from block environment changes by using the same one as on the blockchain.

This contrasts other implementations, where transactions are executed in different block environments than originally, are executed based on a different starting state or ignore the impact of intermediary transactions. All three cases can alter the execution of T_A and T_B, such that the result is not closely related to the blockchain anymore.

3.5.2 Definition weaknesses

An intuitive interpretation of our definition would be, that we compare $T_A \to T_{X_i} \to T_B$ with $T_{X_i} \to T_B$, i.e. reckon what would have happened if T_A was not executed. However, the definition we provide does not perfectly match this concept. Our definition does not consider interactions between T_A and the intermediary transactions T_{X_i}.

In the intuitive model, removal of T_A could also impact the intermediary transactions and thus indirectly change the behaviour of T_B. Then we would not know if T_A directly impacted T_B, or only through some interplay with intermediary transactions. Therefore, excluding the interactions between T_A and T_{X_i} may be desirable, however it can lead to unexpected results if one is not aware of this.

Indirect dependencies

When we analyze a TOD for (T_A, T_B) and there is a TOD between T_A and some intermediary transaction T_X, then removing T_A would impact T_X and thus could indirectly impact T_B.

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 would 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 $(\sigma_{X_n} - \Delta_{T_A})$, we would only modify the balances for a and x, but for b as 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 impact T_B.

3.6 State collisions

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

We define the state collisions of two transactions as:

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

With $W_{T_A} \cap R_{T_B}$ we include write-read collisions, where T_A modifies some state and T_B accesses the same state. With $W_{T_A} \cap W_{T_B}$ we include write-write collisions, where both transactions write to the same state location, for instance to the same storage slot. We do not include $R_{T_A} \cap W_{T_B}$, as we also did not include read-write TOD in our TOD defintion.

3.7 TOD candidates

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

A TOD candidate is not necessarily 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 impacted by this collision.

Conversely, if (T_A, T_B) is TOD, then (T_A, T_B) must also a TOD candidate. For a write-write TOD, this is the case, because both T_A and T_B write to the same state, therefore we have $W_{T_A} \cap W_{T_B} \neq \emptyset$. If we have a write-read TOD, then T_B reads some state that T_A wrote, hence $W_{T_A} \cap R_{T_B} \neq \emptyset$.

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

3.8 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.

3.8.1 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 [Woo24, p.30-38][sml24], we compiled a list of instructions that can access and modify the world state.

In table 3.1 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[Woo24, p.4]. For CALL, CALLCODE and SELFDESTRUCT, this is used to calculate the gas costs. [Woo24, p.37-38] For CREATE and CREATE2, this is used to prevent creating an account at an already active address [Woo24, p.11][1].

In table 3.2 we see instructions that can modify the world state.

3.8.2 Causes without code execution

Some state accesses and modifications are inherent to transaction execution. To pay for the transaction fees, the balance of the sender is accessed and modified. When a transaction transfers some Wei from the sender to the recipient, it als 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. And finally, it also verfies the sender's nonce and increments it by one. [Woo24, p.9]

3.8.3 Relevant collisions for attacks

The previous sections list possible ways to access and modify the world state. Many previous studies have focused on storage and balance collisions, however they did not discuss if or why code and nonce collisions are not important. Here, we try to argue, why

Reference some of them

[1]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 F condition at equation (113).

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

Table 3.1: 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 3.2: Instructions that modify state. A checkmark indicates, that the execution of this instruction can modify this state type.

only storage and balance collisions are relevant for TOD attacks and code and nonce collisions can be neglected.

The idea of an TOD attack is, that an attacker impacts the execution of some transaction T_B, by placing a transaction T_A before it. To have some impact, 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 (vicim) transaction T_B and try to create impactful collisions with a new transaction T_B. We assume some set A to be the set of codes and nonces that T_B accesses and writes.

Let us first focus on the instructions, that could modify the accessed codes and nonces in A, namely `SELFDESTRUCT`, `CREATE` and `CREATE2`. Since the EIP-6780 update[BBF23], 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 use the sender's address for the calculation of the new contract account's address, and both fail when there is already a contract at the target address. [Woo24, p.11] Therefore, we can only modify the code if the contract did not exist previously. If this is the case, it is unlikely that T_B would make a transaction to exactly this attacker-related address. Therefore, none of these instructions is usable for a TOD attack via code or nonce collisions. 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. The only way that T_B can access the nonce of an EOA is through the gas cost calculation when sending Ether to this address. The calculation returns a different cost if the recipient already exists, or has to be newly created. Thus, an attack would be that T_B transfers some Ether to an attacker controlled EOA address a which does not yet exist, and the attacker creates the account at addres a in T_A, which slightly increases the gas cost for T_B. Again, this attack seems negligible.

Therefore, the remaining attack vectors are SSTORE, to modify the storage of an account, and CALL, CALLCODE, SELFDESTRUCT and Ether transfer transactions, to modify the balance of an account.

3.9 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.

How do block rewards play into this? Check and update accordingly.

Theorem 1. *For every transaction T_B after the London upgrade*[2]*, 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. [Woo24, p.8-9]. Therefore, we must have $\sigma(sender)_b \geq v_0$. This requires, that a previous transaction T_A increased the balance of *sender* to be high enough to pay the upfront cost, i.e. $prestate(\Delta_{T_A})(sender)_b < v_0$ and $poststate(\Delta_{T_A})(sender)_b \geq v_0$.

When we calculate $\sigma - \Delta_{T_A}$ for our TOD definition, we would set the balance of *sender* to $prestate(\Delta_{T_A})(sender)_b < v_0$ and then execute T_B based on this state. In this case,

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

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, else we would say that every transaction has been attacked. In the following, we provide some more restrictive definitions.

Frontrunning definitions TBD.

CHAPTER 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 replaying all transactions of a block the same way they were originally executed. With the `prestateTracer` config, this method also outputs, which state has been accessed, and using the `diffMode` config, also which state has been modified[1].

By inspecting the source code from the tracers for Reth[Par24] and results from 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[2]. Therefore, we do not know precisely which 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, giving us a list of collisions. We then use these collisions to obtain a preliminary set of TOD candidates.

[1]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.

[2]I opened a pull request to clarify this behaviour and now this is also reflected in the documentation[noa24b].

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 4.1 and more detailed explanations are in the following sections. The filters are executed in the same order as they are presented in the table and always operate on the output from the previous filter. The only exception is the "Same-value collision" 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 [ZWC$^+$23b]. The filters "Nonce and code collision" and "Indirect dependency" followed directly from our previous theoretical arguments. Further, we also applied an iterative approach, where we searched for TOD candidates in a sample block range and manually analyzed if some of these TOD candidates could be filtered. This led us to the "Same-value collisions" and the "Block validators" filter.

Filter name	Description of filter criteria
Same-value collision	Only take collisions where T_A writes exactly the value, that is read or overwritten by T_B.
Block windows	Drop transactions that 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 TOD candidates with an indirect dependency. e.g. if TOD candidates (T_A, T_X) and (T_X, T_B) exist.
Same senders	Drop if T_A and T_B are from the same sender.
Recipient Ether transfer	Drop if T_B does not execute code.

Table 4.1: 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

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 would have a write-write conflict with all other 99 transactions, the second transaction with the remaining 98 transactions, etc., leading to a total of $\frac{n^2-n}{2} = 4950$ TOD candidates.

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

$$\forall K \in \mathit{collisions}(T_A, T_B)\colon \mathit{poststate}(\Delta_{T_A})(K) = \mathit{prestate}(\Delta_{T_B})(K)$$

With the example of 100 transactions modifying the balance of address a, when the first transaction sets to balance to 1234, it would only have a write-write conflict with transactions where the balance of a was exactly 1234 before the execution. If all transactions wrote different balances, this would 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 indirect 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 $\mathit{poststate}(\Delta_{T_A})(K) \neq \mathit{prestate}(\Delta_{T_B})(K)$, thus some transaction T_X must have modified the state at K between T_A and T_B. Therefore, we would also have a collision (and TOD candidate) between T_A and T_X, and between T_X and T_B. This would be a potential indirect dependency, which could lead to unexpected results as argued in section 3.5.2.

Block windows

According to a study of 24 million transactions from 2019 [ZLYQ21], 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 [Eth24], thus T_A and T_B are at most $\frac{200}{12} \approx 17$ blocks apart from each other. As the study is already 5 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.

Block validators

In Ethereum, each transaction must pay a transaction fee to the block validator and thus modifies the block validator's balance. This would qualify each transaction pair in a block as 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.

Nonce and Code collisions

We showed in section 3.8.3, that nonce and code collisions are not relevant for TOD attacks. Therefore, we ignore collisions for this state type.

Indirect dependency

As argued in section 3.5.2, 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 potentially miss some indirect dependencies. Alternatively, we could also remove a TOD candidate (T_A, T_B) when we also have the TOD candidate (T_A, T_X), however this would remove many more TOD candidates.

Do I want to do this? It would be simpler and more efficient to implement, but we would end up with at most one TOD candidate (T_A, T_B) per transaction T_A. I guess it would be too restricting.

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 $T_A, T_{X_1}, \ldots, T_{X_n}, T_B$ with at least one intermediary node T_{X_i}.

Figure 4.1 shows an example dependency graph, where transaction A influences both X and B and B is influenced by all other transactions. We would filter out the candidate (A, B) as there is a path $A \to X \to B$, but keep (X, B) and (C, B).

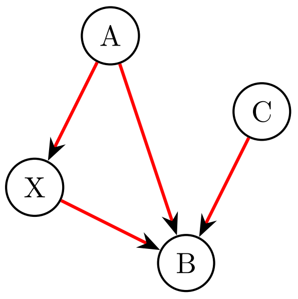

Figure 4.1: TOD candidate filters.

Same sender

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

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

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 can 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 on a randomly sampled sequence of 100 blocks, different to the block range we used for the development of the filters. Refer to chapter 9 for the experiment setup and the reproducible sampling.

We mined the blocks from block 19830547 up to block 19830647, containing a total of 16799 transactions.

4.3.1 Performance

The mining process took a total of 502 seconds, with 311 seconds being used to fetch the data via RPC calls and store it in the database, 6 seconds being 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 can 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 4.2 we can 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 reduced the number of TOD candidates to analyze from more than 60 millions 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 operated on different sets of TOD candidates. Even if "Block windows" would filter out every TOD candidate, this would be less than "Same-value collision" did, because of the order of filter applicatoin.

4.3.3 Transactions

After applying the filters, 7864 transactions are part of at least one TOD candidate. This is, 46.8% of all transactions, that we mark as potentially TOD with some other transaction. Only 2381 of these transactions are part of exactly one TOD candidate. On the other end, there exists one transaction that is part of 22 TOD candidates.

4.3.4 Block distance

In figure 4.2 we can see, that most TOD candidates are within the same block. Morevoer, the further two transactions are apart, the less likely we include them as a TOD candidate.

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 4.2: This table shows the application of all filters used to reduce the number of TOD candidates. Filters were applied from top to bottom and 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.

A reason for this could 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 could be reduced even further without missing many TOD candidates.

4.3.5 Collisions

After applying our filters, we have 8818 storage collisions and 5654 balance collisions remaining. When we analyze, how often each account is part of a collision, we see that collisions are highly concentrated around a small set of accounts. For instance, the five accounts with the most collisions[3] are responsible for 43.0% of all collisions. In total, the collisions occur in only 1472 different account states.

One goal of this paper is to create a diverse set of attacks for our benchmark. 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 more likely related and do not cover a wide range of attack types. To prevent this, we may filter out duplicate addresses for collisions.

Figure 4.3 depicts, how many collisions we would get when we only consider the first n collisions for each address. If we set the limit to one collision per address, we would end up with 1472 collisions, which is exactly the number of unique addresses where collisions happened. When we keep 10 collisions per address, we would get 3964 collisions. Such a scenario would already reduce the amount of collisions by 73%, while still retaining a sample of 10 collisions for each address, that could cover different types of TOD attacks.

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

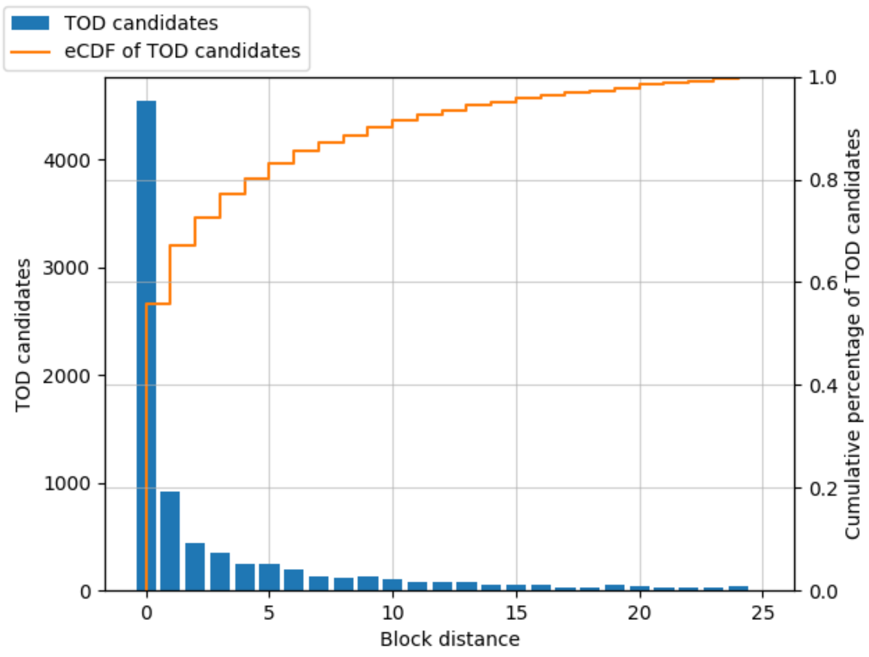

Figure 4.2: The histogram and eCDF of the block distance for TOD candidates. The blue bars show how many TOD candidates have been found, where T_A and T_B are n blocks apart. The orange line shows the percentage of TOD candidates, that are at most n blocks apart.

4.4 Deduplication

TBD.

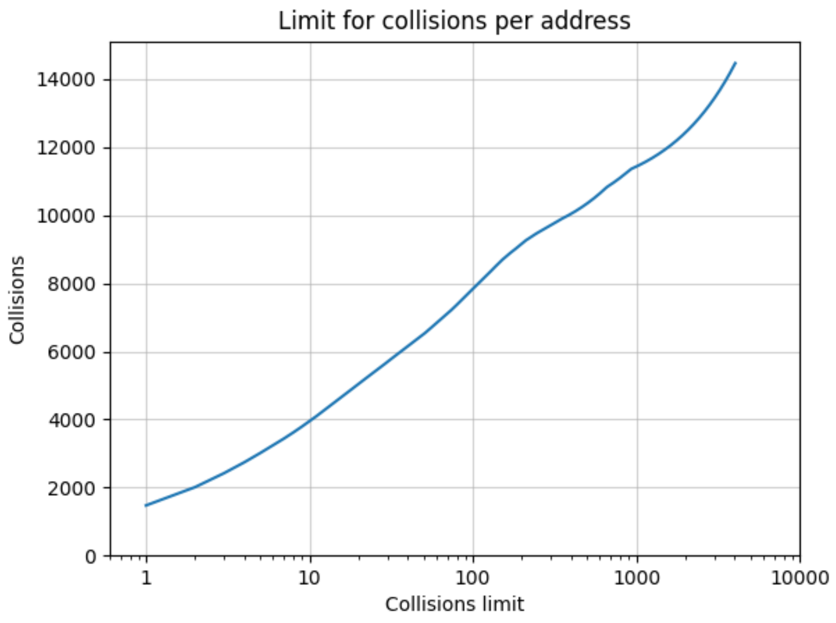

Figure 4.3: The chart shows, how many collisions we have, when we limit the number of collisions we include per address. For instance, if we only include 10 collisions for each address we would end up with about 4000 collisions.

CHAPTER 5

Trace analysis

CHAPTER 6

TOD Attack results

Overall findings of the TOD attack mining and analysis.

CHAPTER 7

Tool benchmarking

7.1 Systematic Literature Review

7.2 Result

CHAPTER 8

Data availability

TBD.

CHAPTER 9

Reproducibility

9.1 Tool

TBD.

9.2 Randomness

TBD.

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 a SN530 NVMe SSD. We used a 16 GB RAM with an additional 16 GB swap file.

For the RPC requests we used a public endpoint[noa24f], which uses Erigon[noa24h] according to the `web3_clientVersion` RPC method. We used a local cache to prevent repeating slow RPC requests. [Fuz24] Unless otherwise noted, the cache was initially empty for experiments that measure the running time.

Overview of Generative AI Tools Used

No generative AI tools where used in the process of researching and writing this thesis.

List of Figures

4.1 TOD candidate filters. 24
4.2 Block distances of TOD candidates 27
4.3 Limit for collisions per address 28

List of Tables

3.1 State accessing instructions 18
3.2 State modifying instructions 18

4.1 TOD candidate filters 22
4.2 Filtered TOD candidates 26

List of Algorithms

Bibliography

[BBF23] Guillaume Ballet, Vitalik Buterin, and Dankrad Feist. EIP-6780: SELFDESTRUCT only in same transaction, 2023. https://eips.ethereum.org/EIPS/eip-6780. Accessed 14.7.2024.

[EMC20] Shayan Eskandari, Seyedehmahsa Moosavi, and Jeremy Clark. SoK: Transparent Dishonesty: Front-Running Attacks on Blockchain. In Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 170–189, Cham, 2020. Springer International Publishing.

[Eth24] Etherscan. Ethereum Average Block Time Chart, 2024. https://etherscan.io/chart/blocktime. Accessed 14.7.2024.

[Fuz24] Fuzzland. ETH RPC Cache Layer, 2024. https://github.com/fuzzland/cached-eth-rpc. Accessed 16.7.2024.

[HKFTW23] Lioba Heimbach, Lucianna Kiffer, Christof Ferreira Torres, and Roger Wattenhofer. Ethereum's Proposer-Builder Separation: Promises and Realities. In *Proceedings of the 2023 ACM on Internet Measurement Conference*, pages 406–420, Montreal QC Canada, 2023. ACM.

[noa23] Gasper, 2023. https://ethereum.org/en/developers/docs/consensus-mechanisms/pos/gasper/. Accesssed 11.7.2024.

[noa24a] Ethereum JSON-RPC Specification, 2024. https://ethereum.github.io/execution-apis/api-documentation/. Accesssed 11.7.2024.

[noa24b] go-ethereum: Built-in tracers, 2024. https://geth.ethereum.org/docs/developers/evm-tracing/built-in-tracers#prestate-tracer. Accessed 14.7.2024.

[noa24c] go-ethereum: debug Namespace, 2024. https://geth.ethereum.org/docs/interacting-with-geth/rpc/ns-debug. Accesssed 11.7.2024.

[noa24d] History and Forks of Ethereum, 2024. https://ethereum.org/en/history/. Accesssed 10.7.2024.

[noa24e] Nodes and clients, 2024. https://ethereum.org/en/developers/docs/nodes-and-clients/. Accesssed 11.7.2024.

[noa24f] POKT Public RPC Endpoints | Nodies DLB, 2024. https://docs.nodies.app/free-endpoints/pokt-public-rpc-endpoints. Accessed 15.7.2024.

[noa24g] Reth Book, 2024. https://reth.rs/jsonrpc/debug.html. Accessed 11.7.2024.

[noa24h] RPC daemon, 2024. https://erigon.gitbook.io/erigon/advanced-usage/rpc-daemon. Accessed 11.7.2024.

[Par24] Paradigm. revm-inspectors, 2024. https://github.com/paradigmxyz/revm-inspectors/tree/b9850ffe4d67aadc46cba5e3798bee459a01a560. Accessed 14.7.2024.

[sml24] smlXL. EVM Codes, 2024. https://www.evm.codes/. Accesssed 10.7.2024.

[TCS21] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner Jones and the Raiders of the Dark Forest: An Empirical Study of Frontrunning on the Ethereum Blockchain. pages 1343–1359, 2021.

[TCS22] Christof Ferreira Torres, Ramiro Camino, and Radu State. Frontrunner Jones, 2022. https://github.com/christoftorres/Frontrunner-Jones/tree/84e98dae4ab37fad7629433fe3ad41967152431f. Accessed 13.7.2024.

[Tik18] Sergei Tikhomirov. Ethereum: state of knowledge and research perspectives. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security: 10th International Symposium (FPS 2017)*, Lecture Notes in Computer Science, pages 206–221. Springer International Publishing, 2018.

[Woo24] Dr Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. Paris version. 2024. https://ethereum.github.io/yellowpaper/paper.pdf. Accesssed 10.7.2024.

[ZLYQ21] Lin Zhang, Brian Lee, Yuhang Ye, and Yuansong Qiao. Evaluation of Ethereum End-to-end Transaction Latency. In *2021 11th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–5, 2021. ISSN: 2157-4960.

[ZWC$^+$23a] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R. Lyu. Combatting Front-Running in Smart Contracts: Attack Mining, Benchmark Construction and Vulnerability Detector Evaluation. *IEEE Transactions on Software Engineering*, 49:3630–3646, 2023. doi: 10.1109/TSE.2023.3270117.

[ZWC$^+$23b] Wuqi Zhang, Lili Wei, Shing-Chi Cheung, Yepang Liu, Shuqing Li, Lu Liu, and Michael R. Lyu. erebus-redgiant, 2023. https://github.com/Troublor/erebus-redgiant/tree/4544163f0c6a369b35c3237851f482d240fa7bbd. Accessed 13.7.2024.