TODLER: A Transaction Ordering Dependency anaLyzER - for Ethereum Smart Contracts

1st Sundas Munir
School of Information Technology
Halmstad University, Sweden
sundas.munir@hh.se

2nd Christoph Reichenbach
Department of Computer Science
Lund University, Sweden
christoph.reichenbach@cs.lth.se

***Abstract*—Smart contracts are programs with data (mutable state); stored on and executed by blockchain platforms. The transactions (or function invocations) dispatched to smart contracts often change their state. In the Ethereum blockchain, nodes (aka miners/validators) can schedule a set of transactions in any order in a block. Multiple transactions in a single block operating on a contract's shared state may yield different outcomes based on their execution order, thus creating a possibility for non-determinism and races between transactions. The resulting issue in Ethereum smart contracts is Transaction Ordering Dependency (TOD). Detecting a TOD requires identifying valid transactions affecting a contract's global/state variables which is equivalent to detecting read-after-write dependencies in race detection, and we expect it to be similarly nontrivial for human developers. In this paper, we identify various TODs, including a novel type previously undocumented in the literature. To detect these TODs, we propose an information flow analysis-based static analyzer, TODler. Our manual evaluation of 108 Ethereum smart contracts shows that TODler outperforms previously available approaches in terms of both run time and precision and also detects the novel TOD pattern identified in this paper.**

***Index Terms*—static analysis, smart contracts, vulnerability detection**

I. Introduction

Smart contracts are computer programs stored on and executed by a decentralized distributed network of peer-to-peer nodes. Smart contracts allow access to their mutable state through interfaces, which can be invoked by many users through transactions [1]. The nodes of the blockchain network maintain a distributed ledger of such transactions that are packaged (in batches) into uniquely identifiable blocks linked together to form a blockchain.

Ethereum [1] is a prominent blockchain platform that supports the execution of smart contracts. Ethereum smart contracts are written in high-level languages (like Solidity or Vyper) and are compiled into low-level EVM bytecode for execution by the Ethereum Virtual Machine (EVM). These smart contracts serve as the backend logic of decentralized applications (Dapps) supported by Ethereum.

In the Ethereum platform, the nodes that create and validate blocks by including a set of transactions are called validators (or miners). All transactions dispatched to Ethereum contracts are first placed in a buffer (transaction or memory pool) called 'mempool' before a validator picks them to schedule for execution in the next block. The order in which these transactions are picked from the mempool has no guarantee [2], [3]. Non-deterministic transaction scheduling creates issues when multiple transactions to the same contract are batched in a single block, and at least one transaction modifies the contract's state variables. Sergey and Hobor [4] describe it as concurrent objects using shared memory. Instead of shared memory, smart contracts have a mutable state stored on the blockchain, accessible by multiple transactions (similar to threads). Although these transactions execute sequentially and atomically, their arbitrary ordering within a block can lead to non-deterministic outcomes. For example, for two transactions, a and b in a single block, if b accesses a contract's storage (global/state variables) affected by a, their execution order can result in two different contract states. As a result, smart contracts can become susceptible to read-write hazards, and assumptions about the global variables of a smart contract could be misleading due to transaction execution races [3]. This paper refers to these races as transactional data races.

Not all transactional data races are harmful to smart contracts. For instance, it is reasonable for the first transaction to receive the reward for submitting the correct solution. However, when transactional data races lead to non-deterministic outcomes that create security vulnerabilities in smart contracts, it is referred to as Transaction Ordering Dependency (TOD), also known as race condition/front-running. The Decentralized Application Security Project (DASP) taxonomy [2] categorizes TOD as one of the top 10 vulnerabilities in Ethereum smart contracts. TOD exploitation attacks involve manipulating the execution order of transactions, as revealed in a real-life incident involving a high-profile Ethereum smart contract [5]. Various tools and frameworks have been created to automate TOD detection. These tools use different analysis approaches and include static analysis-based [6], symbolic execution-based [2], [3], [7], formal verification-based [8], dynamic analysis-based [1], [9], [10], and machine learning-based [11] analyzers. As only 25% of deployed Ethereum smart contracts have their source code available [12], analyzers that take EVM

This research is part of Halmstad University projects funded by Sweden's ELLIIT Strategic Research Environment and was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

[1]The Ethereum blockchain: https://ethereum.org/en/

[2]DASP Top 10 of 2018: https://dasp.co/#item-7

bytecode as input can analyze a broader range of contracts.
Analyzing EVM bytecode requires decompiling it into some intermediate representation so that analysis techniques can be applied. Gigahorse [13], [14] is a decompiler framework and toolchain that converts Ethereum smart contracts from low-level EVM code to a higher-level function-based three-address representation, similar to LLVM IR or Jimple [13]. The Gigahorse toolchain is a collection of analyzers that detect several smart contract vulnerabilities but not TOD.
In this work, we present TODler: a novel security analyzer that takes bytecode as input and performs information flow analysis to detect TOD in Ethereum smart contracts. TODler is built as a client analyzer on top of Gigahorse. TODler specifically tracks information flow to locate memory locations susceptible to read-after-write hazards and uses this information to check if such memory indexes are used in statements that transfer cryptocurrencies (Ethers). The contributions of this paper are as follows:

- We identify transactional data races in Ethereum smart contracts and enumerate different TOD vulnerabilities arising from them (Section II), including a novel TOD pattern that has not been (fully) investigated in the previous literature.
- We implement an automated analyzer, TODler, which statically detects TOD in Ethereum smart contracts (Section III).
- We perform an experimental evaluation of TODler on the dataset of 108 Ethereum contracts that we manually analyze for ground truth (Section IV). We also compare TODler's detection results with comparable state-of-the-art smart contract analyzers, Oyente [2] and Securify [6]. TODler outperforms these analyzers in terms of both run time and precision and is also able to detect a novel TOD pattern.

II. Background

This section presents background details on the transactional data races and multiple forms of TOD vulnerabilities in Ethereum smart contracts.

A. Transactional Data Races

Concurrent systems have long grappled with the problem of data races due to transaction ordering [8]. Solidity does not support concurrency, and transactions dispatched to smart contracts execute sequentially and atomically. However, a node of the blockchain platform can influence the outcome by reordering transactions in a block [8], [15]. The nodes in the Ethereum platform use a consensus mechanism consisting of a complete stack of ideas, protocols, and incentives [16] to reach a common agreement about the next (set of transactions in the) block that will permanently change the state of the blockchain. Ethereum has recently transitioned from a Proof-of-Work (POW) based consensus to a Proof-of-Stake (POS) based mechanism [16]. In a POW-based consensus [16], all nodes (called the miners) compete by solving a difficult math problem, and the winning node gets to create the next block and earn a reward (for example, a certain amount of Ethers) [17]. Alternatively, in a POS-based mechanism [16], a random node called a validator (usually with a higher stake in the network) is assigned to create and validate a new block to be added to the blockchain. In the block creation process, a validator picks a set of transactions from the transaction pool. Ethereum charges a certain fee called "gas" to execute transactions and incentivizes validators to prioritize transactions with a higher gas price. Also, Ethereum bounds each block to utilize a limited amount of gas, permitting only finite transactions in a block. This means including and ordering certain transactions in a block can maximize the profit for the validator in addition to the standard block production reward [18]. In Ethereum, obtaining the maximum profit out of a block is referred to as the Maximal Extractable Value (MEV). In practice, nodes run complex algorithms on blockchain data to detect profitable MEV opportunities and submit those profitable transactions to the network [18]. While it allows the validators to gain maximum profit from each block, MEV on Ethereum can have negative implications. For instance, if a validator observes the upcoming transaction purchasing a large sum of tokens, they can schedule their transaction first to buy the tokens themselves and increase the price for the transaction purchasing those tokens. Also, MEV-related risks are not only related to Ethereum's deprecated POW consensus; Ethereum's transition to POS potentially introduces new MEV-related risks, such as an increase in validator's centralization [18]. Validators are not the only entity responsible for prioritizing the transactions; adversaries can also influence the gas price of transactions, for instance, by providing more gas with their transaction to overrun the victim contract's transaction and thus obtain limited control of transaction execution ordering.

B. TOD and its Forms

Transactional data races can be potentially harmful when the state of smart contracts becomes dependent on the order of transaction execution. This paper focuses on a specific form of transaction ordering dependency (TOD) [2], [6], [7], [19]–[21] which arises when multiple transactions are included in a block, and they read from and write to global state variables of a contract. This makes TOD a Blockchain-level (or consensus/protocol-level) vulnerability [22] because nodes (validators in POS and miners in POW) are responsible for including/executing transactions in an unpredictable order. The execution order of these transactions will result in different contract states depending upon which transaction is executed first. Previous literature, including the studies of Tsankov et al. [6] and Bose et al. [7] target at most three kinds of transaction dependencies, i.e., whether the amount to be transferred, the receiver, or the reachability of the transfer statement (CALL instruction) as a whole are affected due to transaction ordering [23]. The resulting TODs are TOD-Amount(TA), TOD-Recipient (TR), and TOD-Transfer (TT), respectively. In the present work, we specify a new type of TOD that arises from a `selfdestruct` instruction in the contract becoming affected due to transaction ordering. The `selfdestruct`

instruction terminates the contract and transfers the balance of the contract (Ethers) to the recipient specified in its arguments. The recipient's address can be affected due to transaction dependencies when it is directly taken from or is checked against global variables susceptible to read-write hazards. Similarly, checking such global variables in a guarding condition of a `selfdestruct` instruction may also affect its execution. We name this TOD issue TOD-Selfdestruct (TS), and it arises when a `selfdestruct` instruction either transfers Ethers to an arbitrary recipient or itself executes in-deterministically due to transaction dependencies. TS is orthogonal to TR, and TT explored in the previous literature [6], [7] such as TR and TT will miss the occurrences when an explicit transfer of Ethers is not made (for example, by using the CALL instruction), but the contract is terminated using `selfdestruct`. Next, we explain each of these vulnerabilities using abstract examples with simplified Solidity code.

TOD-Amount: If re-ordering a pair of transactions in a block causes a different amount to be transferred, the contract is vulnerable to the TOD-Amount. An Ether transfer statement may load the amount to be transferred from the contract's storage (global state variables). If a contract contains a public function that allows changes to such a value, predicting the transfer amount becomes indeterministic. Listing 1 shows an example of the TOD-Amount in a simplified Solidity contract.

```
contract TODAmount {
   address public owner;
   uint public reward;
   function setReward() public {
      require(msg.sender == owner); //if sender is the owner
      reward = msg.value; //change the reward
   }
   function claimReward(uint256 solution) public {
      require(solution < 10); // if solution is correct
      msg.sender.transfer(reward); //transfer reward to the
      // transaction's sender
   }
}
```

Listing 1. Solidity contract containing TOD-Amount

We assume a scenario: Alice creates a contract (Listing 1) to solve a mathematical problem and sets a reward price for anyone who solves the problem first. Alice can at any time call `setReward()` function to change the reward value. Bob solves the problem and sends the transaction expecting the reward. Suppose Alice decides to reduce the reward and sends the transaction to the contract, and both transactions from Alice and Bob are included in the same block. Now the order in which both of these transactions execute will lead to two different outputs. If Alice sets the gas price higher, her transaction will take precedence, and Bob will get a different reward than expected. If Alice is also a miner/validator, she can get information about the block's upcoming transactions and reduce the reward to zero with a higher gas price transaction to get a free solution.

TOD-Recipient: TOD-Recipient is analogous to the TOD-Amount, whereas the recipient of a transfer becomes susceptible to indeterministic modifications.

```
contract TODRecipient {
   address public owner;
   uint public reward;
   function changeOwner(address _owner) public {
      require(msg.sender == owner); //if sender is the owner
      owner = _owner; //change the owner to new owner
   }
   function withdrawReward() public {
      require(msg.sender == owner);
      owner.transfer(reward); //transfer reward to owner
   }
}
```

Listing 2. Solidity contract containing TOD-Recipient

As an example of a TOD-Recipient, Listing 2 displays a simplified Solidity contract with two public functions; one changes the contract owner, and the other sends a reward amount to the owner. An adversary, in this case, can generate a composite attack by scheduling two transactions in a block where he changes the owner (to himself or another accomplice) and then claims the reward. In a real-life incident of the Parity multi-sig wallet library hack [3], a spelling mistake in the constructor function unintendedly made it a public function allowing changes to the owner variable.

TOD-Transfer: If re-ordering a pair of transactions in a block makes a transfer happen in-deterministically, the contract is vulnerable to TOD-Transfer vulnerability. It is a common practice to validate Ether transfers with conditions, for example, to check whether the caller of the transaction is allowed to make the transfer or if the amount of the transfer is not more than the existing balance of the contract. If the values used in the condition can be updated in other public functions, they create a control dependency for the Ether transfer.

```
contract TODTransfer {
   uint public reward;
   uint256 public secret;
   function initSecret(address _owner) public {
      secret = msg.value; //initialize the secret
   }
   function submitSolution(uint256 guess) public {
      require(guess == secret); //if guessed the correct secret
      msg.sender.transfer(reward); //transfer reward to the
      // transaction's sender
   }
}
```

Listing 3. Solidity contract containing TOD-Transfer

As an example of a TOD-Transfer, Listing 3 displays a simplified Solidity contract with two public functions; one changes the secret, and the other sends a reward to anyone who guesses the secret. We assume two transactions in a block in which Bob guesses the secret and Alice changes the secret. Now Bob may or may not get the reward depending on the order of execution of these transactions.

TOD-Selfdestruct: If a `selfdestruct` instruction either transfers Ethers to an arbitrary recipient or executes in-deterministically due to re-ordering a pair of transactions in a block, the contract is vulnerable to TOD-Selfdestruct (TS). Listing 4 displays a simplified Solidity contract that highlights three ways a TS can arise. The first way is that

[3]https://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/

one transaction changes the owner through `changeOwner` (line 11), and another invokes either `destroy1` (line 18) and `destroy2` (line 22). The second way is that one transaction changes the bidder through `changeLastBidder` (line 13), and another invokes `destroy3` (line 26). Both of these cases transfer Ethers to different recipients depending upon which transaction gets executed. The third way is that one transaction bids through `changeLastBidder` and another invokes `destroy4` (line 30), executing the `selfdestruct` in-deterministically depending upon which transaction gets executed first.

```
1 contract TODSelfdestruct {
2    address public owner;
3    address public lastBidder;
4    uint allBids;
5    constructor(){
6       owner = msg.sender;
7       allBids = 0;
8    }
9    function changeOwner(address _owner) public {
10      require(msg.sender == owner); //if sender is the owner
11      owner = _owner; //change the owner to new owner
12   }
13   function changeLastBidder(uint bidAmnt) public {
14      require(bidAmnt >= 1);
15      allBids += bidAmnt;
16      lastBidder = msg.sender; //change the last bidder
17   }
18   function destroy1() public {
19      require(msg.sender == owner); //if the sender is owner
20      selfdestruct(payable(owner)); //destory the contract
21      //& transfer Ethers (balance) to the owner
22   }
23   function destroy2() public {
24      require(msg.sender == owner); //if the sender is owner
25      selfdestruct(payable(msg.sender)); //destory the
26      //contract & transfer Ethers (balance) to the transaction's sender
27   }
28   function destroy3(address receiver) public {
29      require(receiver == lastBidder);
30      selfdestruct(payable(receiver)); //destory the
31      //contract & transfer Ethers (balance) to the last bidder
32   }
33   function destroy4() public {
34      require(allBids >= 10); //if the bids reach a max threshold
35      selfdestruct(payable(lastBidder)); //destory the
36      //contract & transfer Ethers (balance) to the last bidder
37   }
38 }
```

Listing 4. Solidity contract containing TOD-Selfdestruct

III. Implementation

This section presents the details of the implementation of TODler.

Analysis Pipeline: TODler is implemented in Datalog as a client analyzer on top of the Gigahorse decompiler [13], [14]. The analysis pipeline works as follows:

- The low-level EVM bytecode of the input contract is decompiled into a high-level representation.
- The Gigahorse decompiler constructs the data and control flow dependencies of the EVM bytecode.
- TODler uses derived facts from the decompilation step and formulates Datalog rules, and performs queries using these rules.
- The analysis results are saved in relational files (.csv files) and point out transactions susceptible to specific TODs.

Datalog: Datalog is a declarative specification language that is based on the logic programming paradigm. A Datalog program consists of Horn clauses like facts representing data or (true) knowledge about the world and rules allowing the deduction of facts from other facts [24]. A fact can be created from multiple rules, and a rule can use previously generated facts to infer more facts further. A Datalog rule is of the form:

header(x,y) :- fact1(x,z), fact2(y,z),

where fact1 and fact1 in the body of the rule infer another fact in the header of the rule. In this example, both $(fact1)(x, z)$ and $(fact1)(y, z)$ must hold for specific x, y, and z for the rule to trigger. We express disjunction with the syntax $(\text{fact3}(x); \text{fact4}(x))$: here, only one of $\text{fact3}(x, y)$ or $\text{fact4}(x, z)$ needs to hold.

Decompiler: The Ethereum Virtual Machine (EVM) is a stack-based architecture that executes smart contracts in the EVM bytecode format [25]. Smart contracts written in high-level languages thus need to be compiled into EVM bytecode before they can be executed. The decompilation step reconstructs the bytecode to a high-level representation. The Gigahorse decompiler is based on declarative static program analysis and is implemented using Datalog [14]. It produces functional 3-address IR to write specialized security-related analyses on top of it. Gigahorse offers a Datalog API for the decompilation results and libraries of analysis functions, such as the data-flow analysis library used in TODler [14]. A prime example of the effectiveness of these libraries is their use in the implementation of MadMax [26], which showcases the power of the client analysis infrastructure of Gigahorse.

TODler Rules and Queries: TODler contains several declarative rules which are used to query specific TODs. Specifically, a predicate that we implement to be used by all the queries is `ReadAfterWriteVariables(...)`, which tracks the variables that are susceptible to read-after-write operations. Then we query if these values are used in Ether transfer (for TA, TR, and TT) or `selfdestruct` instructions (for TS). Next, we describe each TOD query in detail.

TOD-Amount: The pattern to detect TOD-Amount checks if the transfer amount is susceptible to a read-after-write operation. The following is the simplified Datalog implementation of TOD-Amount:

```
TODAMOUNT(stmt)←
    READAFTERWRITEVARIABLES(v),
    CALL(stmt,_,_,v,_,_,_,_).        // stmt transfers amount v.
```

TOD-Recipient: This pattern tracks the addresses used as recipients in the Ether transfer statement (CALL) and checks if they are susceptible to a read-after-write operation.

```
TODRECIPIENT(stmt)←
    READAFTERWRITEVARIABLES(v),
    CALL(stmt,_,v,_,_,_,_,_).        // stmt transfers to recipient v.
```

TOD-Transfer: This pattern checks if the values used in the conditions guarding the Ether transfers can be updated in other functions.

```
TODTransfer(gstmt)←
    ControlsWith(gstmt, blk, v),
    CALL(stmt, _, _, _, _, _, _, _),
    StatementBlock(stmt, blk),
    ReadAfterWriteVariables(v),
    StatementUses(gstm, v, _).
```

TOD-Selfdestruct: This pattern combines multiple predicates representing different cases of TS. It checks whether the address used as the recipient in the `selfdestruct` instruction is susceptible to a read-after-write (cf. TS-Case1 in code below) or is checked against a global value susceptible to a read-after-write (cf. TS-Case2). Then it checks whether the values used in the conditions guarding the `selfdestruct` are susceptible to read-after-write operations (cf. TS-Case3).

```
GuardedSelfdestruct(s, g, r)←
    ControlsWith(g, blk, _),      // g guards a blk,
    SELFDESTRUCT(s, r),           // selfdestruct with recipient,
    StatementBlock(s, blk).       // selfdestruct is inside the blk.
// TS-Case1: The recipient of selfdestruct is susceptible to read-after-write.
TODSelfdestructRecipient(s)←
    ReadAfterWriteVariables(r),
    SELFDESTRUCT(s, r).
// TS-Case2: The recipient of selfdestruct is checked against a value
   // that is susceptible to read-after-write.
TODSelfdestructGuardedRecipient(g)←
    GuardedSelfdestruct(_, g, r),
    ReadAfterWriteVariables(v),
    StatementUses(g, v, _),
    (v = r; DataFlows(v, r)).
// TS-Case3: The selfdestruct is guarded against a value that is
   // susceptible to read-after-write.
TODSelfdestructGuarded(g)←
    GuardedSelfdestruct(_, g, _),
    ReadAfterWriteVariables(v),
    StatementUses(g, v, _).
// TS is detected if one of the above cases is true.
TODSelfdestruct(stmt)←
    (TODSelfdestructRecipient(stmt);
    TODSelfdestructGuardedRecipient(stmt);
    TODSelfdestructGuarded(stmt)).
```

IV. Preliminary Evaluation

This section presents details about the dataset used and the results obtained from the experiment to evaluate TODler.

Dataset: The dataset contains 108 vulnerable contracts from the public repository of Gigahorse benchmarks available on Github [4]. We chose this dataset because the source and binary files for each contract are available and can be used to evaluate tools that take any of these formats as input.

Experiment: We executed TODler (and other tools opted for comparison) using an idle machine with an Intel 2.6 GHz CPU with six cores and 16 GB of RAM. Our experiments aim to answer one specific research question:

RQ: How effective is TODler compared to existing (comparable) TOD detection tools?

To find a fitting answer to this question, first, we manually inspected the source files from the dataset and labeled each contract with a specific TOD form if it has a similar code layout as presented in simplified examples in Section II-B. Then, we analyzed these contracts by executing TODler and comparable TOD detection tools.

Comparison against other tools: Tools that support TOD detection include Securify [6], SAILFISH [7], Oyente [2], NPChecker [3], Zeus [8], EthRacer [1], Etherolic [9], Confuzzius [10], and ContractWard [11]. For the preliminary evaluation, we intend to compare TODler with static analysis-based tools only, excluding dynamic analysis and machine learning-based tools (EthRacer, Etherolic, Confuzzius, and ContractWard). Among the existing static analysis-based tools, SAILFISH, ZEUS, and NPChecker are not publicly available. Thusly, we executed the remaining tools, Securify and Oyente, using SmartBug platform [27]–[29] on the source files in the dataset and compared the results with TODler, which takes the (runtime) bytecode of the same contracts.

Securify [6] is an abstract interpreter that encodes dependencies inferred from a contract's control flow graph as logical facts and specifies security properties in terms of compliance and violation patterns using these facts. Violations report all behaviors matching the violation pattern, and warnings report all remaining behaviors not matching the compliance pattern. Securify has two versions, the deprecated Securify [5], and Securify 2.0 [6]. We tested both versions, and this section reports the results of Securify2.0 because the deprecated Securify failed to report any violations or warnings against TT.

Oyente [2] is a symbolic execution-based tool for detecting security issues in Ethereum smart contracts. The core analysis of Oyente checks whether two distinct traces exhibit varying Ether flows. Then it validates those traces by querying Z3 solver [30]. This validation step is reported to be incomplete due to the execution environment of Ethereum not being fully simulated [2]. As a result, Oyente is precise but incomplete as it reduces false warnings but suffers from a high number of false negatives. Oyente only detects TA and TR, and the reported warnings do not explicitly indicate the kind of detected TOD. Therefore, Table I combines Oyente results for TA and TR and does not report anything on TT and TS.

Statistics: Table I shows the number of contracts identified through manual inspection and by executing Securify, Oyente, and TODler for each TOD kind. Based on manual inspection, Table II presents each tool's false positives and negatives for each TOD kind. As, Securify generates violations (V) and warnings (W), Table II reports false positives (FP) for violations or warnings generated by Securify that do not pertain to any actual TOD present in the contracts. Furthermore, if only a warning (without any violation) is generated for a true instance of TOD, we consider it a true positive (TP). Table II displays true positive results for Oyente if it generates a warning for a contract, in which we manually verify the presence of either TA or TR (at the specified code segment). Conversely, if Oyente issues a TOD warning for a contract in which none of its supported TOD forms exist, we classify it as a false positive in Table II. Lastly, false negatives (FN) are all the vulnerable TOD instances in the contracts that the tools missed to detect. Also note that a single contract can be susceptible to multiple TOD forms, which results in more than 108 counts in the tables.

[4]https://github.com/nevillegrech/gigahorse-benchmarks

[5]Deprecated Securify: https://github.com/eth-sri/securify

[6]Securify v2.0: https://github.com/eth-sri/securify2

Reflecting on the Research Question: Lastly, we use the above statistics to answer the research question of whether TODler performs an effective analysis as compared to comparable TOD detection tools. Firstly, TODler detects a novel TOD form, TOD-Selfdestruct, which was not previously targeted in the literature (and by comparable tools, Securify, SAILFISH, and Oyente). It also outperforms Securify and Oyente with regard to execution time (for analyzing the dataset), as follows:

TABLE I
ANALYSIS RESULTS WITH SECURIFY'S (V)IOLATIONS AND (W)ARNINGS.

TOD Form	Manual Inspection	Securify	Oyente	**TODler**
TOD-Amount	21	32(V) + 18(W)	20	12
TOD-Recipient	24	15(V) + 14(W)		25
TOD-Transfer	47	18(V) + 22(W)	-	45
TOD-Selfdestruct	5	-	-	5

TABLE II
MANUAL DETERMINATION OF THE GROUND TRUTH WITH TRUE POSITIVES(TP), FALSE POSITIVE (FP), AND FALSE NEGATIVES(FN).

TOD Form	Securify			Oyente			**TODler**		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
TOD-Amount	17	33	1	13	7	19	9	**3**	12
TOD-Recipient	21	8	1				19	**6**	5
TOD-Transfer	32	7	13	-	-	-	**42**	**3**	**6**
TOD-Selfdestruct	-	-	-	-	-	-	5	0	0
Total	70	48	15	13	7	19	75	12	23

```
Execution time of Securify: 3 hours 8 minutes and 11 seconds
Execution time of Oyente: 8 minutes and 20 seconds
Execution time of TODler: 5 minutes and 36 seconds
```

Secondly, for the TOD forms (TA, TR, and TT) detected by Securify, Table II shows that TODler outperforms Securify by increasing true positives for TT and decreasing false positives for all TOD forms. Although, Securify detects more true instances of TA and TR than TODler, but at the cost of false violations and warnings (cf. Tables II). Through manual inspection, we found that most of these instances use the contract's balance as the amount to be transferred, which the contract did not update in any of the public functions. A contract's balance may unexpectedly change between interleaving transactions due to asynchronous callbacks or if external contracts are called through a `delegatecall`, giving them access to the state of the caller contract [31]. Despite this, the balance is still commonly used as a transfer amount, often reflecting the intentional behavior of the contract. Therefore, issuing TA warnings for using the balance (which the contract itself does not update) as the transfer amount may cause many false alarms. Thirdly, for Oyente's supported TOD forms (TA and TR), TODler comparatively achieves higher true positives and lower false positives and negatives (Table II).

V. LIMITATIONS

In this section, we present the limitations of the current implementation of TODler.

Front Running Attacks: In front-running attacks, a piece of known information is utilized to gain the upper hand. In smart contracts, an attacker can leverage public information and schedule his transaction in a particular order to gain illicit rewards. The implemented TOD patterns in TODler particularly focus on Ether transfer using global variables susceptible to read-after-write operations. The patterns do not detect the (only) reads of global variables because not all globally known values can be utilized to exploit TOD (for instance, the owner is a commonly used global variable to verify a transfer).

Non-Ether transfer TOD forms: TOD can also involve transactions unrelated to asset transfers. Such instances have been investigated as event-ordering bugs, in which changing the order of transactions can result in an unpredictable state of smart contracts [1]. The current implementation of TODler does not detect such TODs.

The possibility of TOD: A possible read-after-write might not actually happen. For example, a write operation might be guarded through conditions, but the patterns do not check that and currently serve as a warning. We plan to cover this limitation in the future by covering the notion of guards, as in the analysis implemented by Lexi et al. [31].

VI. RELATED WORK

The present work is most closely related to studies that address concurrency aspects and non-determinism-related issues of smart contracts [3], [4], [32], and the works proposing methods or tools for TOD detection in Ethereum smart contracts.

Studies on concurrency and non-determinism-related issues: Shuai Wang et al. [3] enumerate various factors that could introduce non-deterministic payment bugs (analogous to the TOD-Amount) in Ethereum smart contracts. That study is closely related to the present work because it addresses non-deterministic transaction scheduling and TOD bugs. That study also considers unpredictable execution of `selfdestruct` due to data races. However, that study does not address specific TOD forms explored in this work, such as TOD-Transfer and specific cases of TOD-Selfdestruct. Sergey and Hobor [4] compare smart contracts with concurrent objects using shared memory. They use this analogy to illustrate issues in smart contracts from the concepts of concurrency theory and motivate using state-of-the-art verification techniques for concurrent programs to smart contracts. That study does not propose any specific approach to detect such issues. In contrast, the present work focuses on detecting specific issues arising from transactional data races in Ethereum smart contracts. Meixun Qu et al. [32] use an analogy similar to Sergey and Hobor [4] to highlight concurrency issues in smart contracts and apply a formal verification-based technique to detect such issues in one Ethereum smart contract. That study addresses vulnerabilities concerning the sequence of two transactions producing different outputs, akin to *generalized*

form the *specific* TOD cases addressed in the present work.

Studies on TOD detection tools: Many studies have proposed tools for TOD detection in Ethereum smart contracts, as presented in Table III. In the table, a filled circle (●) indicates that the tool explicitly checks a specific TOD form, and an empty circle (○) indicates it does not do so. A half-filled circle with a question mark (◑) indicates that the tool possibly detects an issue similar to the specific TOD form but does not explicitly or fully detect it. For example, NPChecker [3] does not handle all three cases covered by the TS pattern. NPChecker [3] is a static analysis-based analyzer that detects non-deterministic payment bugs arising from unpredictable transaction scheduling and external callee behavior. NPChecker applies information flow/taint analysis to pinpoint global variables affected by read-write hazards and their influence on funds transfer. NPChecker takes bytecode as input and is the only analyzer other than TODler which takes the execution of `selfdestruct` into account. However, the payment bugs targeted by NPChecker are not *identical* to TOD forms TR, TT, and TS. Securify [6] is a static analysis-based security scanner implemented using Java and stratified Datalog [33]. It has predefined compliance and violation patterns that mirror the security properties' satisfaction or negation. Securify checks three of the four TOD issues TA, TR, and TT detected by TODler. The original version of Securify that takes bytecode as input is deprecated, and the latest Securify 2.0 takes Solidity code as input. In contrast, TODler takes (runtime) bytecode as input, allowing it to detect a comparatively large number of deployed contracts on the blockchain. SAILFISH [7] applies a hybrid approach of static analysis (built on top of the Slither [34] framework) and symbolic evaluation to detect state-inconsistency bugs in Ethereum smart contracts. Similar to Securify, these bugs reflect TA, TR, and TT. Oyente [2] is a symbolic execution-based tool that detects TOD occurrences if a contract transfers Ether differently when the order of transactions changes. Compared with the TODler, Oyente takes the Solidity code of Ethereum smart contracts as input and does not detect the TT and TS issues. ZEUS [8] is an abstract interpretation and symbolic model checking-based framework for detecting security issues in smart contracts, including TOD. ZEUS catches TOD by detecting writes and subsequent reads to global variables across a pair of transactions using policy specifications for correctness and fairness criteria. Zeus does not directly take into account the TOD forms targeted by TODler. Similar to the core idea behind NPChecker, Zeus determines potential read-write hazards for global variables that can influence Ether flows, for instance, if such variables are used in Ether transfer statements. This excludes TR and TT *implicitly*, and TS *explicitly*.

Apart from static detection of TOD before deployment, many studies have proposed using dynamic analysis. Such as, EthRacer [1] combines symbolic execution of contract events with fuzzing of event sequences and detects TOD by checking whether changing the ordering of function invocations results in differing outputs. Such issues are event ordering bugs which are a generalized form of TOD types detected by TODler. CONFUZZIUS [10] also leverages fuzzing and constraint solving to generate sequences of transactions causing TOD in Ethereum contracts. CONFUZZIUS detects whether two transaction execution traces read from and read to the same storage variable of a contract, implying that it may detect (some of the) specific TOD forms addressed in the present work. Etherolic [9] performs a runtime analysis of Ethereum contracts using concolic testing and dynamic taint tracking to detect TOD, referred to as race conditions. The authors [9] do not specify the kinds of TOD detected by Etherolic; therefore, Table III highlights them as a possibility.

Lastly, machine learning algorithms have also been applied to detect TOD in Ethereum smart contracts before deployment. ContractWard [11] is one such tool that applies multiple supervised classification algorithms to detect TOD. Since ContractWard does not target specific forms of TOD, Table III highlights a possibility against TODs addressed in this work.

VII. Conclusions

This paper investigates data races affecting smart contracts due to the arbitrary scheduling of transactions by the nodes of the Ethereum blockchain. When such transactional data races cause indeterministic state outcomes in smart contracts, the resulting class of vulnerabilities is called Transaction Ordering Dependency (TOD). We present a static analysis-based tool, TODler, which detects TOD in Ethereum smart contracts. TODler applies information flow analysis to track read-after-write possibilities to state variables used in Ether transfers and `selfdestruct` instructions and conditions guarding these statements. Our experimental evaluation reported that TODler outperforms state-of-the-art TOD detection tools Securify and Oyente in terms of both run time and precision and also detects the novel TOD pattern (TS) that we specify in this paper. This work serves as a stepping stone for our future endeavors in detecting concurrency and non-determinism-related issues in smart contracts.

VIII. Data Availability

The dataset used for evaluation contains 108 smart contracts from 'vulnerable-bytecode' of the public repository on GitHub [7]. The tool produced in this research is also publicly available [8].

References

[1] 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*, 2019, pp. 363–373.

[2] 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*, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 254–269. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[3] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–29, 2019.

[7] https://github.com/nevillegrech/gigahorse-benchmarks

[8] Todler artifact: https://doi.org/10.5281/zenodo.7740758

TABLE III
A COMPARISON WITH TOD DETECTION TOOLS. INPUT{**B**YTE**C**ODE, **S**OLIDTY**C**ODE},{● YES, ○ NO, ◑ POSSIBILE BUT DOES NOT DIRECTLY APPLY.}

Tool	Analysis Approach	Input	TOD Forms			
			TA	TR	TT	TS
TODler	Static analysis	BC	●	●	●	●
NPChecker [3]	Static analysis	BC	●	◑	◑	◑
Securify [6]	Static analysis	SC	●	●	●	○
SAILFISH [7]	Hybrid (Static analysis & Symbolic evaluation)	SC	●	●	●	○
Oyente [2]	Symbolic execution	SC	●	●	○	○
Zeus [8]	Abstract interpretation & symbolic model checking	SC	●	◑	◑	○
EthRacer [1]	Fuzzing & Symbolic execution	BC	◑	◑	◑	◑
Confuzzius [10]	Hybrid fuzzing	SC	◑	◑	◑	◑
Etherolic [9]	Concolic testing & dynamic taint tracking	BC	◑	◑	◑	◑
ContractWard [11]	Machine learning	SC	◑	◑	◑	◑

[4] I. Sergey and A. Hobor, "A concurrent perspective on smart contracts," in *International Conference on Financial Cryptography and Data Security*. Springer, 2017, pp. 478–493.

[5] SECBIT, "How the winner got fomo3d prize — a detailed explanation," https://medium.com/coinmonks/how-the-winner-got-fomo3d-prize-a-detailed-explanation-b30a69b7813f, Aug 2018.

[6] P. Tsankov, A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli, and M. Vechev, "Securify: Practical security analysis of smart contracts," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 67–82.

[7] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 161–178.

[8] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: Analyzing safety of smart contracts," in *NDSS*, 2018.

[9] M. Ashouri, "Etherolic: a practical security analyzer for smart contracts," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020, pp. 353–356.

[10] C. F. Torres, A. K. Iannillo, A. Gervais, and R. State, "Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021, pp. 103–119.

[11] W. Wang, J. Song, G. Xu, Y. Li, H. Wang, and C. Su, "Contractward: Automated vulnerability detection models for ethereum smart contracts," *IEEE Transactions on Network Science and Engineering*, vol. 8, no. 2, pp. 1133–1144, 2020.

[12] S. Lagouvardos, N. Grech, I. Tsatiris, and Y. Smaragdakis, "Precise static modeling of ethereum "memory"," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–26, 2020.

[13] N. Grech, L. Brent, B. Scholz, and Y. Smaragdakis, "Gigahorse: thorough, declarative decompilation of smart contracts," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1176–1186.

[14] N. Grech, "Gigahorse toolchain," https://github.com/nevillegrech/gigahorse-toolchain, 2021.

[15] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F.-Y. Wang, "An overview of smart contract: architecture, applications, and future trends," in *2018 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2018, pp. 108–113.

[16] E. Docs, "Consensus mechanisms," https://ethereum.org/en/developers/docs/consensus-mechanisms/, December 2022, accessed: 2023-01-11.

[17] D. Mingxiao, M. Xiaofeng, Z. Zhe, W. Xiangwei, and C. Qijun, "A review on consensus algorithm of blockchain," in *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, 2017, pp. 2567–2572.

[18] E. Docs, "Maximal extractable value (mev)," https://ethereum.org/en/developers/docs/mev/#miner-extractable-value, November 2022, accessed: 2023-01-10.

[19] S. Munir and W. Taha, "Pre-deployment analysis of smart contracts – a survey," 2023. [Online]. Available: https://arxiv.org/abs/2301.06079

[20] N. Atzei, M. Bartoletti, and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," in *International conference on principles of security and trust*. Springer, 2017, pp. 164–186.

[21] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*, 2018, pp. 9–16.

[22] A. Dika and M. Nowostawski, "Security vulnerabilities in ethereum smart contracts," in *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018, pp. 955–962.

[23] C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei, "ethor: Practical and provably sound static analysis of ethereum smart contracts," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 621–640.

[24] S. Ceri, G. Gottlob, L. Tanca *et al.*, "What you always wanted to know about datalog(and never dared to ask)," *IEEE transactions on knowledge and data engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[25] G. Wood *et al.*, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, vol. 151, no. 2014, pp. 1–32, 2014.

[26] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," vol. 2, no. OOPSLA, Oct. 2018. [Online]. Available: https://doi.org/10.1145/3276486

[27] J. F. Ferreira and P. Cruz, "Smartbugs: A framework for analysing ethereum smart contracts," https://github.com/smartbugs/smartbugs, 2023.

[28] T. Durieux, J. F. Ferreira, R. Abreu, and P. Cruz, "Empirical review of automated analysis tools on 47,587 ethereum smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 530–541.

[29] J. F. Ferreira, P. Cruz, T. Durieux, and R. Abreu, "Smartbugs: A framework to analyze solidity smart contracts," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1349–1352.

[30] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[31] L. Brent, N. Grech, S. Lagouvardos, B. Scholz, and Y. Smaragdakis, "Ethainter: a smart contract security analyzer for composite vulnerabilities." in *PLDI*, 2020, pp. 454–469.

[32] M. Qu, X. Huang, X. Chen, Y. Wang, X. Ma, and D. Liu, "Formal verification of smart contracts from the perspective of concurrency," in *International Conference on Smart Blockchain*. Springer, 2018, pp. 32–43.

[33] D. U. Jeffrey, "Principles of database and knowledge_base systems, volume i," 1988.

[34] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 2019, pp. 8–15.