TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts

Chenyang Ma
School of Computer Sci. & Eng.
Nanjing University of Sci.& Tech.
Nanjing, China
15189826258@163.com

Wei Song*
School of Computer Sci. & Eng.
Nanjing University of Sci. & Tech.
Nanjing, China
wsong@njust.edu.cn

Jeff Huang
Parasol Laboratory
Texas A&M University
College Station, TX, USA
jeff@cse.tamu.edu

ABSTRACT

Smart contracts are programs that define rules for transactions running on blockchains. Since any qualified transaction sequence within the same block can be orchestrated by the blockchain miner, unexpected results may occur due to data races between transactions (called transaction races). Surprisingly, transaction races in smart contracts have not been fully investigated. To address this, we propose TransRacer, an automated approach and open-source tool that employs symbolic execution to detect transaction races in smart contracts. TransRacer analyzes function dependencies to identify transaction races hidden in specific contract states. It also generates witness transactions that can trigger such races. The experimental results on 50 real-world smart contracts show the effectiveness and efficiency of TransRacer: it detects 426 races in 255.9 minutes, including 149 race bugs leading to inconsistent states.

CCS CONCEPTS

• **Security and privacy** → *Domain-specific security and privacy architectures*; • **Software engineering** → *Software defect analysis.*

KEYWORDS

Ethereum, smart contract, data race, symbolic execution

ACM Reference Format:
Chenyang Ma, Wei Song, and Jeff Huang. 2023. TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616281

1 INTRODUCTION

Blockchain is a powerful infrastructure that enables mutually untrusted users to reach a consensus on a distributed ledger. The ledger can be used to decentralize applications, which are known as smart contracts [3]. Conceptually, a smart contract is a stateful program that can be invoked by external users via transactions. A smart contract manages transaction executions through the pre-defined rules written in a Turing-complete language (e.g., Solidity [7]).

This paper investigates a specific type of vulnerability in smart contracts caused by their concurrent execution model. As demonstrated in [30], for transactions submitted to Ethereum over a brief time period, the outcome may vary due to the nondeterministic execution sequence of the transactions determined by the miner. Prior studies have emphasized the seriousness of this issue. Luu et al. [22] introduce the transaction-ordering-dependent (TOD) bug to explain how a pair of transactions can exhibit different contract balances under different execution sequences. Kolluri et al. [16] extend the concept of this concurrent bug to the event-order bug, which captures the full-state differences of smart contracts beyond the balance difference. We find that state differences or inconsistent states occur mainly because transactions race to access the same variables in a smart contract. This situation is called a *transaction race* (race for short) in this paper. When two transactions executed in different sequences generate different contract states, they lead to a *transaction race bug* (race bug for short). The inconsistent states impact the subsequent transactions. What is worse, an attacker can tamper with some storage variables that transitively impact on money transactions [2]; let alone the cases when the race bugs directly lead inconsistent balances and money loss. Groce et al. [12] show that roughly 41% of race conditions examined have serious legal and financial implications. However, no tools have been designed to detect races for smart contracts.

To this end, we propose TransRacer, an efficient symbolic analysis tool for race detection. The goal of TransRacer is to find a reachable contract state and two transactions for each candidate function pair such that the transactions exhibit races at that state. A race requires that the two transactions have read/write conflicts and can be executed in different orders. Moreover, whether the transactions can be executed depends on the state of the global variables. TransRacer utilizes *symbolic execution* to capture all the above conditions and employs *constraint solving* to check the satisfiability of conditions. When a race is detected, TransRacer further checks whether the race could result in a race bug. TransRacer aims to help developers/auditors in the process of testing their implementation against their model or intended behaviour. The detected races/race bugs are malignant if contracts are not designed deliberately; otherwise, they are benign. TransRacer can be used for contract auditing before contracts are deployed to Ethereum. Users can only provide contract bytecode to TransRacer.

Since smart contracts are stateful programs, certain functions can be successfully invoked only at specific contract states. Thus, symbolic analysis for race detection cannot proceed without inferring such states. Moreover, it is not easy to find such a state for each

*Corresponding author.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616281

of these functions, as the number of reachable contract states is enormous. To tackle this challenge, TransRacer leverages *function dependencies* to identify functions whose execution can cause a function not initially callable to become callable at the updated state. Existing smart contract bug detectors [14, 22, 25, 34] usually restrict vulnerability detection to a specific contract state or explore different contract states with the help of fuzzing, but none attempt to infer such states like TransRacer.

For efficiency reasons, TransRacer takes the following steps. First, TransRacer statically prunes function pairs that share no read/write variables. Second, at each contract state that activates a specific function, TransRacer checks only the function pairs that contain this function during race detection. Therefore, TransRacer captures races at multiple contract states but does not perform excessively repetitious checking of the same function pairs.

In this study, we implement the TransRacer symbolic analysis tool in Python and evaluate the effectiveness and efficiency of TransRacer on 50 randomly chosen smart contracts. TransRacer successfully finds 426 true races, including 149 race bugs, while requiring an average of only 5.1 minutes of analysis time per contract. We further apply TransRacer to investigate races in 6,943 real-world smart contracts, and the empirical results reveal that 53.0% (3,680/6,943) of smart contracts can lead to races, of which 73.6% (2,710/3,680) can lead to race bugs. These results indicate that races and race bugs are prevalent in practice.

The main contributions of our work are as follows:

(1) TransRacer can effectively detect races between two transactions to the same smart contract, without generating false positives. It also provides witness transaction sequences to manifest the races, thus reducing manual auditing efforts.
(2) Instead of relying on random fuzzing, we propose to seek contract states that activate certain functions by analyzing the dependence relations between functions, which significantly shrinks the race detection search space. Our proposed notion of *function dependence* and the approach to seek the function dependence are applicable to detect other contract vulnerabilities.
(3) We design and implement an open-source tool, TransRacer, that automatically detects races in smart contracts. An experimental evaluation on 50 real-world smart contracts show the effectiveness and efficiency of TransRacer.
(4) We apply TransRacer to 6,943 real-world smart contracts. The results indicate that races are a severe problem that are potentially harmful in practice.

The rest of this paper is organized as follows. Section 2 presents an example that motivates our work. Section 4 and Section 5 describes and evaluates our approach, respectively. Section 6 reviews related work, and Section 7 concludes the paper.

2 MOTIVATION

In this section, we use an example to motivate our work. With this example, we illustrate how transaction races can lead to severe outcomes (e.g., attacks, money loss) and how to detect them.

Figure 1(a) depicts a snippet of the smart contract *WinToken* that is compliant with the Ethereum Request for Comments 20 (ERC20) token standard [5]. The function `mint()` is used to create

```
1  contract WinToken{
2   uint256 public totalSupply;
3   mapping(address => uint256) balances;
4   mapping (address => mapping (address => uint256))
        internal allowed;
5
6   function setTransferLock(bool _set) onlyOwner {
7    lockTransfer = _set;}
8   function mint(address _to, uint256 _amount)
        onlyOwner {
9    totalSupply = totalSupply.add(_amount);
10   balances[_to] = balances[_to].add(_amount);}
11  function approve(address _spender, uint256 _value){
12   allowed[msg.sender][_spender] = _value;}
13  function increaseApproval(address _spender, uint
        _addedValue){
14   allowed[msg.sender][_spender] = allowed[msg.sender
         ][_spender].add(_addedValue);}
15  function transferFrom(address _from, address _to,
        uint256 _value){
16   require(lockTransfer == false);
17   require(_to != address(0));
18   require(_value <= balances[_from]);
19   require(_value <= allowed[_from][msg.sender]);
20   allowed[_from][msg.sender] = allowed[_from][msg.
       sender].sub(_value);...;}...}
```

(a)

Sequences	
$\texttt{mint}_{\texttt{owner}}$(A, 500) $\texttt{increaseApproval}_{\texttt{A}}$(S, 500)	
$\texttt{approve}_{\texttt{A}}$(S, 200) $\texttt{transferFrom}_{\texttt{S}}$(A, B, 200)	$\texttt{transferFrom}_{\texttt{S}}$(A, B, 200) $\texttt{approve}_{\texttt{A}}$(S, 200)
Result 1	**Result 2**
S has transferred 200 tokens to B and can transfer no more	S has transferred 200 tokens to B and can transfer 200 more

(b)

Figure 1: *WinToken* contract: (a) the contract fragment; (b) the transaction sequences that trigger a race.

tokens. The functions `increaseApproval()` and `approve()` serve the same purpose of approving others to spend a certain number of tokens (by calling `transferFrom()`) from the token owner. The invocations of the functions `approve()` and `transferFrom()` both race to access the same variable *allowed*. Figure 1(b) demonstrates that the race bug between $approve_{\mathrm{A}}(S, 200)$[1] and $transferFrom_{\mathrm{S}}(A, B, 200)$ emerges only after $mint_{\mathrm{owner}}(A, 500)$ and $increaseApproval_{\mathrm{A}}(S, 500)$ have been executed. The reason lies in the fact that the two conditions at Lines 18-19 of `transferFrom()` can be satisfied only after both `mint()` and `increaseApproval()` are invoked. That is, the execution of `transferFrom()` depends on the successful executions of `mint()` and `increaseApproval()`. This inspires us to analyze *function dependencies* (cf. Definition 2) to detect races and race bugs hidden at specific contract states.

3 PROBLEM FORMULATION

A *smart contract* c is a five-tuple $c = \langle address, b, V, F, s_0 \rangle$, where *address* is the contract address, which is used for identifying the smart contract; b is the contract balance; V is the set of storage variables; F is the set of contract functions; and $s_0 : \{b\} \cup V \rightarrow \mathbb{R}^n$ is the initial contract state, which maps b and the variables in V to the concrete values in $\mathbb{R}$. When a smart contract is deployed, it is configured to an initial state. The contract state can be changed by

[1] A represents the sender of this transaction.

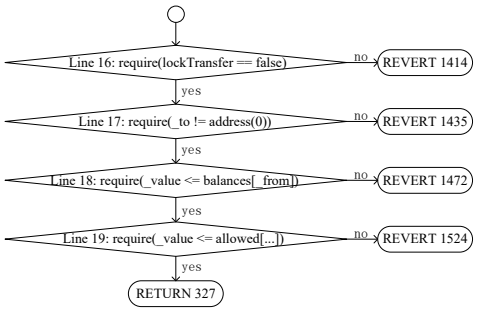

Figure 2: Control flow graph of the function `transferFrom()`.

sending transactions to it. A transaction sent to c is a concrete call to a function in c. A transaction t consists of a sender and a receiver, the ether value associated with t, the function $f \in F$ invoked by t, and the data input to f. The ether value is paid by the transaction sender for the transaction, e.g., tokens.

A *valid path* of a function f is one that leads to a normal outcome (e.g., RETURN or STOP), while an *invalid path* for f is one that results in an exception (e.g., REVERT or INVALID). The path that a transaction selects for execution depends not only on the transaction itself but also on the contract state. Figure 2 shows a control flow graph for the function `transferFrom()` shown in Figure 1, in which the nodes labeled with RETURN and REVERT represent the ends of valid and invalid paths, respectively. The path set of a function f consists of the valid path set (P_v) and the invalid path set (P_i). The integer value after RETURN/REVERT represents the position of the RETURN/REVERT operator in the smart contract bytecode.

If path p executed by transaction t is a valid path, the contract state changes from s to s' (denoted as $s \xrightarrow{t} s'$); otherwise, the contract state remains unchanged [20]. At s, a path of a function f is *feasible* if there exists a transaction that can execute this path successfully. For instance, at a contract state where $lockTransfer$ is true, only the path that ends at REVERT 1414 is feasible. A function is *callable* if at least one valid path of this function is feasible. A race between two functions requires that both functions are callable and exhibit read/write conflicts if invoked in different orders.

Definition 1 (**Transaction Race**/**Transaction Race Bug**). *Given a smart contract $c = \langle address, b, V, F, s_0 \rangle$, a transaction race will occur between $f_1 \in F$ and $f_2 \in F$ if there exists a contract state s at which two transactions t_1 and t_2 invoke f_1 and f_2, respectively, such that the following conditions hold:*

(1) $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2$ *and* $s \xrightarrow{t_2} s_1' \xrightarrow{t_1} s_2'$.
(2) $(R_{t_1} \cap W_{t_2}) \cup (W_{t_1} \cap R_{t_2}) \cup (W_{t_1} \cap W_{t_2}) \neq \emptyset$.

where R_{t_1}/W_{t_1} and R_{t_2}/W_{t_2} are the sets of storage variables that are read/written by t_1 and t_2, respectively. A transaction race can lead to a transaction race bug if $s_2 \neq s_2'$.

Transaction race bugs can lead to different contract states when the execution order of the two transactions is reversed. Thus, these bugs share something in common with event-order bugs [16]. The difference is that the former involves only two transactions at a time, while the latter considers multiple transactions. However, after studying their experimental results [16], we find that all their

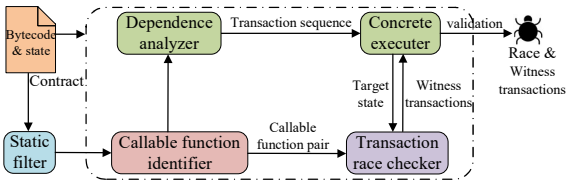

Figure 3: The framework of TransRacer.

reported event-order bugs are caused by two transactions. The other transactions are only used to update the contract states at which the event-order bugs can occur. Thus, the number of bugs/races involving multiple transactions is small in practice. Therefore, in this paper, we only check races/race bugs between two transactions. TOD bugs [22] are also caused by two transactions but are relevant only to balance differences, whereas transaction race bugs are more general because both balance differences and contract storage differences are considered.

The research problem studied in this paper is as follows: *Given a smart contact at state s_0, for each pair of functions f_1 and f_2 where read/write conflicts can occur, we aim to determine whether there is a reachable state s (a transaction sequence σ such that $s_0 \xrightarrow{\sigma} s$) on which there are two transactions t_1 and t_2 calling f_1 and f_2, respectively, such that a race or a race bug occurs between t_1 and t_2.*

One may argue that techniques such as fuzzing or symbolic execution can be used to find such reachable contract states. However, the application of these techniques without further optimization may cause transaction-sequence explosion problems. To this end, we utilize symbolic execution to efficiently find the qualified reachable contract state.

4 TRANSRACER

The framework of TransRacer is presented in Figure 3. The input consists of the bytecode and the initial state s_0 of a smart contract, which can be obtained directly from Ethereum. The output consists of the function pairs that may lead to races (race bugs) and witness transactions that can trigger the races (race bugs). Given a smart contract with n functions, the number of candidate function pairs that may lead to races is $n + C_n^2 - m$, where n and C_n^2 represent the number of function pairs composed of two identical functions and two different functions, respectively, and m represents the number of function pairs that do not share read/write variables.

In the framework, the component *static filter* serves as a preprocessing step to extract the function pairs with shared read/write variables. Since this component is implemented in [16], we directly adopt their implementation. The *static filter* first symbolically executes a function f and monitors the data flowing into the opcodes SLOAD/SSTORE and SHA3 to obtain the possible read/write variables in each path of f. Based on these results, it compiles the function pairs that may access the same storage variables. For the two functions of each candidate function pair, the component *callable function identifier* identifies whether both are callable. The component *transaction race checker* employs symbolic execution to detect races between two functions that are both callable at state s. Note that a function f may become callable at different contract states. Thus, we aim to check races for f at a set of states, where

each state activates at least one valid path of f. For the functions that involve at least one infeasible valid path, the component *dependence analyzer* first utilizes symbolic execution to determine the function dependencies; then, based on these function dependencies, for each infeasible valid path p_i, it determines transactions that cause the contract to enter a target state at which p_i becomes feasible. For a function pair (f_1, f_2), the *dependence analyzer* seeks two transaction sequence sets T_{f_1} and T_{f_2} that activate the paths in f_1 and f_2, respectively. Then, based on T_{f_1} and T_{f_2}, a set of contract states, S, is obtained for race checking. Each state $s' \in S$ is obtained by the concrete execution of a transaction sequence $\sigma_1 \in T_{f_1}$ and a transaction sequence $\sigma_2 \in T_{f_2}$. When the race checker checks races for f_1 and f_2 at the contract state s', it only explores the paths that are activated by σ_1 and σ_2. In this way, we avoid to repetitively verify of the same paths at different contract states. Although we cannot guarantee that hidden races can be revealed at the target state, we believe our heuristic is feasible for balancing race detection effectiveness and efficiency. Finally, the *concrete executer* executes the witness transactions to validate the correctness of the races.

4.1 Transaction Race Checking

In this section, we introduce how to determine whether a function is callable at a given contract state s and how to find races for a pair of callable functions at state s.

Callable function determination. At any contract state s, for a function f, we can symbolically execute f to obtain a path p of f and the corresponding path constraints Φ_p. By constraint solving based on a satisfiability modulo theories (SMT) solver, we can determine whether Φ_p can be satisfied. When Φ_p can be satisfied, p is a feasible path, and if f has a feasible valid path, it is callable.

Race detection. As illustrated in Definition 1, three conditions are required to trigger transaction race bugs. We construct *path constraints*, *read/write constraints*, and *distinct post-state constraints* to capture these conditions.

(1) Path constraints. Given a smart contract with two functions f_1 and f_2 and a contract state s, the symbolic analysis takes four steps to find two transactions t_1 and t_2 that can invoke f_1 and f_2 in different orders: (i) we symbolically execute f_1 at s to seek a valid path p_1 and mark the symbolic state after p_1 as s_1; (ii) we symbolically execute f_2 at s_1 to seek a valid path p_2; (iii) we symbolically execute f_2 at s to seek a valid path p'_1 and mark the symbolic state after p'_1 as s'_1; (iv) we symbolically execute f_1 at s'_1 to seek a valid path p'_2. The path constraints collected during the four stages can be denoted as Φ_{p_1}, Φ_{p_2}, $\Phi_{p'_1}$, and $\Phi_{p'_2}$. If the path constraints $\Phi_{\text{path}} := \Phi_{p_1} \wedge \Phi_{p_2} \wedge \Phi_{p'_1} \wedge \Phi_{p'_2}$ are satisfied, then f_1 and f_2 can be invoked in different orders. Our symbolic engine will branch if it meets an "if" condition or an exit point of a loop during path exploration. We set 750 as the maximum number of explored branches to limit the number of iterations of loops.

(2) Read/write constraints. The read/write constraints are used to determine read/write conflicts. We identify the read/write storage locations by inspecting the opcodes `SLOAD/SSTORE` and `AND`. This approach is reasonable as the Ethereum virtual machine (EVM) usually reads/writes a variable with the cooperation of these two opcodes: `SLOAD/SSTORE` is used to load/store the data with a size of 256 bits, and `AND` is used to select the target storage location from the 256 bits. After identifying the storage locations accessed by the two functions during the symbolic analysis, we obtain the read/write constraints Φ_{rw} based on condition (2) of Definition 1.

(3) Distinct post-state constraints. The distinct post-state constraints Φ_{dist} are constructed to ensure that the output (the values written to the storage variables) differs between the two functions executed in different orders.

We claim that a race (race bug) is detected only when the race constraints $\Phi_{\text{race}} := \Phi_{\text{path}} \wedge \Phi_{\text{rw}}$ (the race bug constraints $\Phi_{\text{bug}} := \Phi_{\text{race}} \wedge \Phi_{\text{dist}}$) are satisfied. Then, the witness transactions can be extracted directly from the solution returned by the SMT solver.

For a function pair that may be subject to races, with the help of the *static filter*, we safely prune the path pairs that do not share any read/write variables. We note that the execution of two function paths can read/write the same variable, but it does not indicate that they can access the same storage location. For instance, in Figure 1, functions `mint()` and `transferFrom()` write the same variable *balances*. The function `mint()` can write the storage location of *balances*[*0*], but function `transferFrom()` cannot write this storage location because of the constraint in Line 17. Therefore, it is necessary to construct the read/write constraints to ensure the read/write conflict.

It is worth mentioning that when more than two functions share read/write variables, our approach handles each pair of functions individually. The reasons are two-fold. First, two functions sharing read/write variables are the smallest unit for a race. From the perspective of debugging and auditing, we need to know whether such two functions can lead to a race. Second, as the number of functions sharing read/write variables increases, verifying different invocation orders of the functions will result in a combinatorial explosion problem, which is more time-consuming.

4.2 Function Dependence Analysis

As previously mentioned, certain function paths can only be triggered successfully at specific contract states. For a given function f, we try to trigger each valid path of f. In the following part, we show how to find a contract state to trigger a valid path. The other valid paths are triggered in the same way.

Since the only way to change the contract state is to submit transactions to the contract, for a path $p_v \in P_v$ that cannot be triggered at the initial contract state s_0, we aim to find a transaction sequence $\sigma = t_1 t_2 ... t_m$ such that $s_0 \xrightarrow{\sigma} s'$ and p_v becomes feasible at s'. We leverage *function dependencies* (cf. Definition 2) to find such transactions one by one. One transaction in σ can be obtained (by querying the SMT solver) once a function dependence is determined.

Definition 2 (**Function Dependence**). *For two functions f and f_1, we say that f is function-dependent on f_1 if and only if there are two transactions t_{f_1} and t_f which invoke f_1 and f, respectively, such that the following two conditions hold:*

(1) $s \xrightarrow{t_1} s_1$.
(2) *The path p_f executed by t_f at state s_1 cannot be executed by t_f at state s.*

Example 1. *In Figure 1, the path (referred to as p_v) ends at* `RETURN` *327 is infeasible initially. Hence, the execution of transferFrom$_S$(A, B, 200) (referred to as t) ends at* `REVERT` *1472. However, after mint$_{\text{owner}}$(A,*

Table 1: Symbols Used in Algorithms 1~2

Symbol	Description
P_f	The set of feasible valid paths of function f
$jump\_num(p, p_v)$	The number of joint JUMPI instructions between p and p_v
$max\_jump\_num(p_v)$	The maximum number of joint JUMPI instructions between any feasible path and p_v
$FD(f_i, f)$	Whether f is function-dependent on f_i

500) (referred to as t_1) is executed, the tokens held by A change to 500 (i.e., condition (1) in Definition 2 is satisfied). At the updated contract state, the execution of transferFrom$_S$(A, B, 200) can proceed to REVERT *1524 (referred to as p'). In such case, condition (2) in Definition 2 is satisfied. Thus,* `transferFrom()` *is function-dependent on* `mint()`.

Since a function f may depend on one or more functions, we propose Heuristic 1 to determine all these functions.

Heuristic 1. *If a path $p_v \in P_v$ of a function f is infeasible at a contract state s, a function on which f depends is first determined and then concretely executed, updating the contract state to s_1. These steps are iterated until a reachable contract state s' is obtained at which p_v becomes feasible.*

If heuristic 1 works, a transaction sequence $\sigma = t_1 t_2 ... t_n$ is found such that $s \xrightarrow{\sigma} s'$. At s', we use the method discussed in Section 4.1 to check whether a race occurs between the two callable functions.

The intuition behind Heuristic 1 is explained as follows. For a path $p = b_1 ... b_n$ of a function f that cannot be executed when f is called, where b_i $(1 \le i \le n)$ represents a block of opcode, several conditions should be satisfied to make p fully executed. When we find a function f_1 on which f depends, the corresponding transaction which calls f_1 can make such a condition satisfied; thus, the call of f can go ahead, say, reaching b_m, where $m < n$. We iteratively find a function on which f depends and make the function concretely execute until b_n is reached when f is called. With Heuristic 1, we aim to efficiently find a transaction sequence to activate p. Otherwise, we have to enumerate all transaction sequences until σ is found.

Algorithm 1 searches for transaction sequences to some states at which the valid paths of a function become feasible, where Table 1 summarizes the symbols used that are not defined in the main text. Lines 1~6 activate the valid paths of a function f. Lines 2~3 trigger the valid paths of f by invoking function *seq_generation*(). Lines 4~5 update the feasible valid paths and their triggering transaction sequences and states. Algorithm 1 stops once all infeasible valid paths of f have been checked (Line 6).

At the opcode level, the number of paths of a function could be extremely large. For example, the contract *CityToken* has over 250,000 paths in the control flow graph of the opcode. Fortunately, we observe that these paths share some common opcode blocks, and thus when a path p is found infeasible in Algorithm 1, those paths with p as the prefix are also known to be infeasible. With this observation, Algorithm 1 mitigates the path exploration problem to a large extent. Moreover, the subsequent race checking are only conducted on these activated valid paths instead of all paths, and thus the efficiency is ensured.

Algorithm 1: Search for transaction sequences to the states where the valid paths of a function become feasible

Input: s - contract state, f - function, $F_f = \{f_1, ..., f_k\}$ - the set of functions that may write variables f reads, *SEQ_LENGTH* - maximum transaction-sequence length.

Output: $\Sigma = \{(\sigma', s', p) \mid s \xrightarrow{\sigma'} s' \wedge p$ is feasible at $s'\}$, where $p \in P_v$.

1 $\Sigma \leftarrow \emptyset$, $s' \leftarrow s$, **global** feasible valid path set P
2 **for** *each infeasible valid path p_v of f, $\exists\, p'_v \in P_{f'}$ such that p'_v and p_v may have read/write conflict* **do**
3 $\sigma, s' \leftarrow seq\_generation(F, f, s', p_v)$
4 **if** *p_v becomes feasible at s'* **then**
5 $\Sigma \leftarrow \Sigma \cup \{(\sigma, s', p_v)\}$, $P_f \leftarrow P_f \cup \{p_v\}$
6 **return** Σ
7 **Function** $seq\_generation(F_f, f, s', p_v, SEQ\_LENGTH)$
8 $\sigma \leftarrow \emptyset$
9 **for** $l \leftarrow 1$, $l \le SEQ\_LENGTH$, l++ **do**
10 *dependence_found* $\leftarrow$ False
11 **for** *each $f_i \in F_f$* **do**
12 $t_{f_i}, p_f, FD(f_i, f) \leftarrow dep\_analysis(f_i, f, s', p_v)$
13 **if** *$FD(f_i, f) = True$* **then**
14 *dependence_found* $\leftarrow$ True
15 **if** *$p_f = p_v$* **then**
16 $s' \leftarrow concrete\_execution(s', t_{f_i})$, $\sigma \leftarrow append(\sigma, t_{f_i})$
17 **return** σ, s'
18 **if** *dependence_found* **then**
19 $s' \leftarrow concrete\_execution(s', t_{f_i})$, $\sigma \leftarrow append(\sigma, t_{f_i})$
20 **else**
21 **break**
22 **return** null, null
23 **Function** $dep\_analysis(f_i, f, s', p_v)$
24 $t_{f_i} \leftarrow \emptyset$, $p_f \leftarrow \emptyset$, $FD(f_i, f) \leftarrow$ False, **global** $max\_jump\_num(p_v)$, P_{f_i}
25 **for** *each $p_{f_i} \in P_{f_i}$ that may write the variables p_v reads* **do**
26 $s_1, \Phi_1 \leftarrow symbolic\_execution(s', p_{f_i})$
27 **for** *each path $p \in f$ such that $jump\_num(p, p_v) > max\_jump\_num(p_v)$* **do**
28 $s_2, \Phi_2 \leftarrow symbolic\_execution(s_1, p)$
29 $s_3, \Phi_3 \leftarrow symbolic\_execution(s', p)$
30 **if** *$\Phi_1 \wedge \Phi_2$ is satisfiable and Φ_3 is unsatisfiable* **then**
31 $t_{f_i} \leftarrow solve(\Phi_1 \wedge \Phi_2)$, $p_f \leftarrow p$, $FD(f_i, f) \leftarrow$ True
32 $max\_jump\_num(p_v) \leftarrow jump\_num(p, p_v)$
33 **if** *$p = p_v$* **then**
34 **return** $t_{f_i}, p_f, FD(f_i, f)$
35 **return** $t_{f_i}, p_f, FD(f_i, f)$

Function *seq_generation*() in Algorithm 1 implements Heuristic 1 to trigger a valid path of f. The input includes the contract state s, a valid path p_v of a function f that cannot be executed at s, the set of functions $F_f = \{f_1, ..., f_k\}$ that may write variables read by f, and the specified maximum transaction-sequence length, *SEQ_LENGTH*. The output consists of the found transaction sequence σ and the target state s'. *seq_generation*() iterates the following procedure until either a transaction sequence is returned or the number of iterations reaches the threshold (*SEQ_LENGTH*). We first iterate over the functions $f_1, ..., f_k$ to find the functions on which f depends (Lines 11~12). The function *dep_analysis*() then checks whether $FD(f_i, f)$ is True and ensures that the newly found path p_f can go closely to p_v than previously found paths do. For a function f_i, if $FD(f_i, f)$ is True (Line 13), we check whether p_f is p_v (Line 15). If so, we return the updated σ and s' (Lines 16~17). After all the k function pairs are analyzed, if a function dependence is found (Line

18), we know that although p_v is still infeasible, the invocation of f can go closely to p_v after t_{f_i} is executed. Thus, we concretely execute t_{f_i} to update the contract state and append t_{f_i} to σ (Line 19). Otherwise, *seq_generation*() returns null (Line 22). The time complexity of Algorithm 1 is $O(m^3 \times k \times SEQ\_LENGTH)$, where m is the average number of feasible valid paths of a function, and k is the number of functions in F_f.

Function *dep_analysis*() works as follows. At contract state s', Lines 25~26 select a feasible valid path p_{f_i} of f_i and symbolically execute p_{f_i} to obtain the updated contract state s_1. Lines 28~29 obtain the constraints Φ_2 and Φ_3 that make path p of f feasible at states s_1 and s', respectively. Line 30 checks the satisfiability of the constraints to ensure that $FD(f_i, f)$ holds. Line 32 updates $max\_jump\_num(p_v)$ to ensure that the execution of p_v can proceed.

Example 2 (Continuation of Example 1). *At the initial contract state, p_v is infeasible. First, we find that f depends on* `mint()`. *Note that f also depends on* `setTransferLock()`, *as the execution of $setTransferLock_{owner}(True)$ can cause the execution of t to end at* `REVERT` *1414 (referred to as p''). Algorithm 1 does not consider this function dependence because the number of the joint* `JUMPI` *instructions between p_v and p' is larger than that between p_v and p''. Next, Algorithm 1 concretely executes t_1 to update the contract state. At the new state, Algorithm 1 further finds that f depends on* `increaseApproval()` *and that the execution of t after $increaseApproval_A(S, 500)$ (referred to as t_2) reaches* `RETURN` *327. As a consequence, after $\sigma = t_1 t_2$ is executed, p_v becomes feasible. Finally, the race bug in Figure 1(b) is detected.*

Consider three functions h, g, and f, where f depends on g, and g depends on h. In such case, we say f *nestedly-dependent* on g and h. At the initial contract state, if a valid path of f is infeasible, Algorithm 1 only checks whether there are function dependencies between g and f and between h and f. Since no dependence is found, Algorithm 1 cannot proceed. In fact, a function f may depend on several functions, and some of the functions cannot be activated either at the initial state. To activate f, such functions should be activated first. With this in mind, we propose Algorithm 2 which invokes Algorithm 1 by considering nested function dependencies. Note that we set a threshold *DEPTH_NEST*, and if the depth of the nested function dependence is beyond this threshold, we give up activating the corresponding function f.

Algorithm 2 works as follows. Line 2 only considers the nested function dependencies whose depths are not beyond *DEPTH_NEST*. At each candidate contract state $s \in S$ (Line 6), Algorithm 2 invokes Algorithm 1 (Line 8) for each function $f \in F'$ (Line 7). Next, it updates the feasible valid paths, the functions whose paths still need triggering, and the reachable contract states to further call Algorithm 1 (Lines 9~13). If all functions in F' are handled or no reachable candidate contract state can be found (Line 14), Algorithm 2 returns (Line 16); otherwise, Algorithm 2 iterates until the number of iterations reaches *DEPTH_NEST*. The time complexity of Algorithm 2 is $O(m^4 \times k \times SEQ\_LENGTH \times n^2 \times DEPTH\_NEST)$, where n is the number of functions in F'.

Example 3. *We use contract DAVToken (cf. Figure 4) to show how Algorithm 2 works. In this contract, there is a nested function dependence, because* `transferFrom()` *depends on* `approve()`, *and*

Algorithm 2: Activate valid paths of functions in a contract

Input: s_0 - Initial contract state, F' - the set of functions whose paths need triggering, *DEPTH_NEST* - For any function $f \in F'$, the threshold of the depth of nested function dependence we consider.

Output: $\Sigma = \{(\sigma', s', p) \mid s \xrightarrow{\sigma'} s' \wedge p \text{ is feasible at } s'\}$, where $p \in P_v$.

```
1  S ← {s_0}, Σ ← ∅
2  for each f ∈ F' do
3      P_f ← ∅
4  for l ← 1, l ≤ DEPTH_NEST, l++ do
5      S' ← ∅
6      for each s ∈ S do
7          for each f ∈ F' do
8              Σ' ← call Algorithm 1 for f at state s
9              if Σ' ≠ ∅ then
10                 Σ ← Σ ∪ Σ'
11                 for each (σ', s', p) ∈ Σ' do
12                     S' ← S' ∪ {s'}
13     S ← S'
14     if F' = ∅ ∨ S = ∅ then
15         break
16 return Σ
```

```
1  contract DAVToken{{
2     contractPaused=True;
3     function unpause() public whenNotPaused {
4        contractPaused = False;}
5     function approve(address _spender, uint256 _value)
          {
6        require(!contractPaused)
7        allowed[msg.sender][_spender] = _value;}
8     function transferFrom(address _from, address _to,
          uint256 _value){
9        require(_value <= allowed[_from][msg.sender]);
10       require(...);}
11    ...}
```

Figure 4: A smart contract *DAVToken*.

`approve()` *depends on* `unpause()`. *At the initial state s_0, Algorithm 2 first finds that* `approve()` *depends on* `unpause()`, *while the function dependence between* `approve()` *and* `transferFrom()` *is not found. After* `unpause()` *is concretely executed, a new state s' is obtained. In the next iteration, Algorithm 2 finds that* `transferFrom()` *depends on* `approve()`. *This example shows the limitation of Algorithm 1 due to nested function dependencies, and how Algorithm 2 addresses it.*

4.3 Concrete Execution

The *concrete executer* is used for two purposes. First, it helps analyze function dependencies. At a contract state s, after a function dependence between f_1 and f is found, the *concrete executer* submits a transaction t_1 to call f_1 with the concrete input returned by the symbolic dependence analysis. After the execution of t_1, a new contract state s' is obtained, i.e., $s \xrightarrow{t_1} s'$. Without this state transition, f is not callable, and another function dependence cannot be found. Second, the *concrete executer* verifies the correctness of the race detection results. For a detected race, the *concrete executer* executes the witness transactions in different orders and checks whether a read/write conflict exists. To verify the correctness of detected race bugs, the *concrete executer* further checks whether reversing the execution order produces different contract states.

5 EVALUATION

In this section, we evaluate the effectiveness and efficiency of TransRacer on 50 smart contracts. Then, we report the results of a large-scale empirical study on 6,943 real-world smart contracts. This evaluation aims to answer the following research questions:

- **RQ1**: How effective is TransRacer? Is the *dependence analyzer* helpful in finding races?
- **RQ2**: How efficient is TransRacer? Compared with single-state detection, how much additional time does race detection consume due to the multiple states reached by executing interdependent functions?
- **RQ3**: Are races and race bugs prevalent in practice?
- **RQ4**: What are the numbers and percentages of races hidden at specific contract states (that is, those that cannot be detected at the initial contract states)?
- **RQ5**: What are the numbers and percentages of races caused by calls to the same function (TR_I) and to two different functions (TR_D), respectively?
- **RQ6**: What are the consequences of races?
- **RQ7**: Regarding TR_I and TR_D, which is more likely to cause race bugs and even monetary losses?

Experimental setup. Our experiments are executed on a desktop PC equipped with a Windows 10 operating system, 32 GB of memory, and an Intel i7 9700 CPU. We use the Web3 suite and an agency account provided by Infura[2] to interact with the Ethereum mainnet. The symbolic execution engine is implemented by ourselves and the SMT solver used is z3. The maximum analysis time for one contract is set to 120 minutes. The maximum time allowed for checking races between one function pair is set to two minutes. The nested dependence threshold is set to three because we did not find any cases exceeding this threshold in our experiments.

The maximum transaction-sequence length to activate a valid path of a function is set to four. This is because the experimental results of ETHRacer on 6,943 real-world smart contracts show that only 0.24% contracts involve event-order bugs each of which is triggered by a transaction sequence whose length is greater than four. In our method, since the maximum length of transaction sequence to activate a path of a function is four, the maximum length of transaction sequence to trigger a race is 4+4+2 = 10. Thus, we believe our setting is sufficient for most contracts.

5.1 Experimental Evaluation

We first conducted an experiment to answer RQ1 and RQ2.

Comparison approaches. The approach most closely related to our work is ETHRacer [16], which is a fuzzing-based approach that detects event-order bugs. The input fed to its fuzzer is generated through symbolic analysis. ETHRacer aims to find a set of transactions that exhibit output differences under varying transaction sequences, which shares some commonality with TransRacer. Thus, we incorporate ETHRacer for comparison. Furthermore, we consider three additional tools (Oyente [22], Securify [35], and Sailfish [2]) that detect TOD bugs (i.e., specific race bugs related to contract balances). Oyente is a symbolic analysis tool that aims to detect ether flow differences, while Securify and Sailfish are static analysis tools that can detect bugs that are relevant to inconsistent contract states, such as TOD bugs. The default configurations of the four tools are used. We realize that there are other tools that perform multi-state bug detection, such as Smartian [6] and Teether [17]; however, none of them detect transaction races.

Benchmark. The experiments are performed on 50 real-world smart contracts whose source code is available (the source code facilitates manual confirmation of the reported races). These open-source smart contracts are all randomly selected from the UR dataset created by Ren et al. [27] for empirical study. The transaction counts of the 50 read-world smart contracts are all greater than 100. We do not use the MI [28] and SmartBugs [8] datasets because those contracts only contain a limited number of manually injected bug patterns and there are no false positives that can be generated by different approaches from them. We obtain the initial states of these 50 smart contracts from Ethereum mainnet for the experiments.

Table 2 shows the experimental results on the 50 contracts. The columns "#Func", "#Inst", and "#Dep" represent the number of functions, the number of instructions, and the number of detected function dependencies in each smart contract, respectively. The columns "#TR_I" ("#TRB_I") and "#TR_D" ("#TRB_D") show the number of transaction races (transaction race bugs) caused by transaction calls to two identical or two different functions, respectively. The columns "IS (initial state) " and "US (updated state)" list the number of races that can be triggered at the initial and updated contract states, respectively. IS is the initial state when we begin race detection, which is obtained from Ethereum mainnet. The column "Coverage Rate" lists the ratio of the number of function pairs checked by the *transaction race checker* to the number of function pairs that are not filtered out by the *static filter*. In the sub-columns "#TRB" of the three competitors, the entry with the format of "#TP/#FP" represents the number of true positives and false positives of the approach. For instance, at the row of contract "COW", the entry "0/1" in the sub-column "#TRB" of Oyente represents that it finds zero true positive and one false positive transaction race bug.

We only list TRB_D (transaction race bugs between two different functions) for ETHRacer because it is the only bug type that ETHRacer detects. Instead of actively seeking contract states that trigger race bugs, ETHRacer tests different contract states through fuzzing with longer traces. Thus, we classify the race bugs (also event-order bugs) reported by ETHRacer into a category of bugs detected at the updated states when the trace length is larger than two. We only list #TRB detected by Oyente, Securify, and Sailfish, as they do not find the contract states to trigger the bugs.

Result validation. We manually check the results of these tools to determine whether they are true positives. To examine the correctness of the races detected by TransRacer, we manually inspect the contract source code to ensure that the two functions indeed share common variables and that read/write conflicts over these variables actually exist. The outcome of race bugs is that they can produce inconsistent contract states. The race bugs detected by TransRacer are all true positives because the *concrete executer* ensures that these races can cause inconsistent contract states. For ETHRacer, we execute the witness transactions provided by ETHRacer to verify that they can cause inconsistent contract states. As static analysis tools, Oyente, Securify, and Sailfish only indicate the functions that could potentially cause race bugs. For

[2]https://infura.io

Table 2: Results of Different Race Detectors on 50 Real-world Smart Contracts

Contract Name	#Func	#Inst	#Dep	TransRacer										ETHRacer			Oyente		Securify		Sailfish	
				$\#TR_I$		$\#TR_D$		$\#TRB_I$		$\#TRB_D$		Time	Coverage	$\#TRB_D$		Time	#TRB	Time	#TRB	Time	#TRB	Time
				IS	US	IS	US	IS	US	IS	US	(min.)	Rate	IS	US	(min.)		(min.)		(min.)		(min.)
XCTCrowdSale	5	472	0	0	0	1	0	0	0	1	0	0.6	6/6=100.0%	0	0	2.8	1	0.1	1	0.1	1	0.1
BitcoinBlue	9	939	2	1	2	0	1	1	0	0	0	1.7	7/7=100.0%	0	0	3.8	0	0.1	0	0.1	0	0.1
RADIUM	10	1,426	0	1	0	0	0	0	0	0	0	0.2	5/5=100%	0	0	2.5	0	0.2	0	0.1	0	0.1
BMUS	11	1,600	2	1	2	0	2	1	0	0	1	0.9	7/7=100.0%	0	0	3.6	0	0.1	0	0.1	0	0.1
RippleAlpha	11	2,132	3	2	3	1	6	1	0	1	2	5.5	14/14=100.0%	0	0	6.8	0	0.5	0	0.1	0	0.1
PlayCash	12	2,060	3	2	3	0	8	1	0	0	2	1.9	21/21=100.0%	0	0	9.4	0	0.4	0	0.1	0	0.1
Xpense	12	2,136	3	2	3	0	6	1	0	0	2	2.1	21/21=100.0%	0	0	8.4	0	0.4	0	0.1	0	0.1
BB	12	1579	4	1	4	0	7	1	0	0	2	2.4	12/12=100.0%	0	0	6.7	0	0.2	0	0.1	0	0.1
WEBN	13	1,398	3	2	3	1	6	1	0	1	3	5.9	14/14=100.0%	0	0	4.1	0	0.3	0	0.1	0	0.1
CelebrityMarket	13	1,938	1	0	0	0	0	0	0	0	0	0.3	2/32=6.2%	0	0	7.8	0	0.1	0	0.1	0	0.1
NinjaKittyUnit	14	1,887	0	1	0	0	0	1	0	0	0	0.7	2/10=20.0%	0	0	2.9	0	0.1	0	0.1	0	0.1
COW	15	2,501	1	4	1	3	4	1	0	0	1	3.6	29/29=100.0%	0	0	13.9	0/1	0.1	0/1	0.2	0	0.1
UNTY	16	3,127	0	2	0	0	0	1	0	0	0	1.3	5/7=71.4%	0	0	5.7	0	0.1	0	0.2	0	0.1
ChangeBank	16	1,929	3	1	2	0	1	0	0	0	0	1.0	12/22=54.5%	0	0	7.5	0	0.8	0/1	0.1	0/1	0.1
Aavio	16	2,110	3	3	3	1	7	1	0	1	1	4.8	24/24=100%	1	1	15.8	0	0.9	0	0.1	0	0.1
Freedom	17	2,578	1	2	1	2	1	1	0	2	0	2.7	27/43=62.7%	2	0	50.1	0	0.1	0/2	0.2	0/3	0.1
ETJ	17	3,035	3	4	3	1	8	2	0	0	0	7.5	32/32=100.0%	0	0	37.6	0	0.9	0	0.1	0	0.1
HubrisOne	18	2,924	2	2	1	1	3	1	0	1	2	8.7	28/30=93.3%	0	0	12.5	0	0.6	0	0.1	0	0.1
MADANA	18	3,543	2	2	1	1	2	1	0	1	1	2.8	18/28=64.3%	0	0	62.1	0	0.1	0	0.1	0	0.1
GOG	19	3,304	7	2	7	0	8	1	0	0	2	9.1	64/64=100.0%	0	0	25.1	0	0.5	0	0.1	0	0.1
MediBloc	19	3,237	4	2	1	1	2	1	0	1	2	5.4	78/78=100.0%	0	0	24.6	0	0.9	0	0.3	0	0.1
Simmitri	19	1,885	4	3	3	1	8	1	0	1	2	8.2	26/26=100.0%	1	1	10.0	0	0.1	0	0.1	0	0.1
Winsshar	20	2,504	1	6	1	8	5	2	0	0	0	9.6	57/72=79.2%	0	0	41.8	0	0.1	0	0.1	0	0.1
ProofOfReview	20	1,941	3	2	2	0	2	2	0	0	1	1.5	6/8=75.0%	0	0	3.9	0	0.1	0	0.2	0	0.1
HSD	21	3,345	4	2	3	0	6	1	1	0	1	5.2	43/50=86.0%	0	0	87.9	0	0.4	0/1	0.1	0	0.1
LendConnect	22	2,461	0	1	0	0	0	1	0	0	0	2.5	4/14=28.5%	0	0	35.8	0	0.3	0/1	0.1	0/1	0.4
ChickenFarmer	22	1,731	1	0	0	0	0	0	0	0	0	3.6	6/20=30.0%	0	0	10.1	0	0.1	1	0.3	1/4	0.9
EnchantedShop	23	3,312	2	0	0	0	0	0	0	0	0	2.3	13/25=52.0%	0	0	46.9	0	0.6	1/1	0.2	1/2	0.3
MATOX	23	2,692	3	2	2	0	4	2	0	1	0	9.8	54/60=90.0%	0	0	86.4	0	0.2	1	0.1	0	0.1
Dragon	23	3,238	4	1	4	0	9	1	0	0	1	4.8	16/32=50.0%	0	0	20.3	0	0.1	0	0.1	0	0.1
EthernetCash	23	2,577	3	5	3	1	10	3	0	0	1	9.2	53/53=100.0%	0	0	16.6	1	0.3	1	0.1	1/3	0.1
OMPxContract	24	3,479	1	2	1	0	1	2	0	0	0	1.1	20/52=38.5%	0	0	34.2	0	0.1	0	0.1	0	0.1
Sota	24	2,712	5	3	3	1	2	2	0	0	0	14.0	73/80=91.2%	0	0	97.4	0	0.2	0	0.2	0	0.1
Viewly	24	2,510	0	4	0	0	1	4	0	0	1	1.2	36/86=41.9%	0	0	34.9	1	0.4	2	0.2	0	0.1
Crowdsale	25	2,665	1	0	0	0	0	0	0	0	0	1.6	2/18=11.1%	0	0	14.3	0	0.4	0/2	0.1	0/2	0.6
Char	25	3,653	3	6	2	1	2	4	0	1	2	8.0	52/52=100.0%	0	0	27.4	0	0.3	0	0.1	0	0.1
CityToken	25	3,918	4	2	0	0	2	2	0	0	2	3.5	37/37=100.0%	0	0	11.0	0	0.1	2	0.2	2	0.5
ROD	26	3,539	6	5	4	2	6	3	0	2	4	11.0	144/144=100.0%	0	0	83.9	0	0.9	0	0.1	0	0.1
BrownChipMain	26	3,707	0	2	0	0	0	1	0	0	0	0.5	4/36=11.0%	0	0	16.5	0	0.7	0	0.1	1/4	4.5
grip	27	3,728	4	4	2	0	6	3	0	0	1	15.9	59/66=89.3%	0	0	86.2	0	0.3	1	0.1	0	0.1
TokensWarContract	27	4,722	3	2	1	0	1	2	1	0	1	1.9	42/42=100.0%	0	0	16.4	1	0.3	2	0.2	2/1	0.3
Dentacoin	29	3,322	2	7	2	2	1	6	0	0	1	4.5	48/64=75.0%	0	1	23.0	0	0.7	0	0.1	0/3	0.6
LAAR	29	4,937	7	2	5	1	3	1	0	1	3	9.1	94/112=87.5%	0	0	65.1	0	0.1	0	0.1	0	0.1
INRD	30	4,701	2	4	1	1	2	3	0	1	1	6.3	78/118=66.1%	1	1	47.7	0	0.1	0	0.1	0	0.1
UniondaoDollarToken	31	4,926	0	5	0	0	0	5	0	0	0	2.3	24/36=66.7%	0	0	79.8	0	0.6	0/1	0.2	0/1	1.0
Genaro	34	6,990	0	2	0	0	0	2	0	0	0	1.0	32/40=80.0%	0	0	14.3	0	0.1	0	0.2	0	0.1
CSTK_CLT	34	4,842	6	4	5	1	7	3	0	0	1	6.9	73/111=65.8%	0	0	75.2	0	3.1	0	0.4	0	0.1
Yihaa	37	4,626	3	2	2	0	2	2	0	0	1	3.1	6/8=75.0%	0	0	5.3	0	0.4	0	0.1	0	0.1
IgfContract	37	3,886	7	4	4	2	16	3	0	1	2	21.8	68/108=62.9%	0	0	62.7	0	0.1	2	0.1	0	0.1
Scale	41	4,214	10	3	4	1	7	2	0	1	3	16.4	28/90=31.1%	0	0	84.3	0	0.1	0/1	0.1	0	0.1
Total	1,064	148,618	130	122	94	35	175	81	2	16	50	255.9	/	5	4	1,553.0	4/1	18.5	14/11	7.1	9/25	13.2
Average	21.3	2,972.4	2.6	2.4	1.9	0.7	3.5	1.6	0.1	0.3	1.0	5.1	75.1%	0.1	0.1	31.1	0.1/0.1	0.4	0.3/0.2	0.1	0.2/0.5	0.3

these race bugs, we first deploy the corresponding contracts to a private chain. Then, based on the contract source code and their detection results, we carefully generate and execute transactions to determine whether they can cause inconsistent contract states.

Effectiveness. In the 50 smart contracts, TransRacer finds 426 races, including 149 race bugs. Among the 426 races and 149 race bugs, our manual analysis confirms that all are true positives. For the 50 contracts, there are 187 functions that are not callable at the initial contract state. Heuristic 1 successfully activates 69.5% (130/187) of them.

ETHRacer flags nine event-order bugs for the 50 smart contracts. Our manual analysis confirms that all of the detected bugs are true positives and are detected by TransRacer as well. In other words, TransRacer finds all the bugs reported by ETHRacer but detect 57 additional true positive TRB_D. A close investigation shows that there are two main reasons why ETHRacer finds fewer bugs. First, ETHRacer cannot easily generate witness transactions because it does not infer the three constraints that trigger races. Second, ETHRacer does not infer the contract states that enable race bugs to emerge. Consider the example illustrated in Figure 1. The hidden race can only be triggered at specific contract states. Since ETHRacer does not infer such states, the probability of generating the transactions that trigger this race is small.

As shown in Table 2, Oyente finds five race bugs, four of which are true positives that can be detected by TransRacer. Securify and Sailfish find 25 and 34 race bugs, respectively. We manually check these results and confirm that Securify and Sailfish find 14 true positives and 9 true positives, respectively. Of these true positives, TransRacer misses four. One of them is relevant to the transfer of the ERC-721 token. We cannot generate the input (i.e., token identifier) that triggers this race because the token that can be transferred is unobtainable solely through the contract bytecode. TransRacer

```
1 contract UniondaoDollarToken {
2  address public owner;
3  function withdrawToken(address token, uint256
       amount) {
4     if (token == address(0x0)){
5        owner.transfer(amount);}...}}
```

Figure 5: A false positive reported by Securify.

fails to detect the other three true positives because the distinct post-state constraints of these races involve nonlinear constraints that cannot be solved by the SMT solver. TransRacer misses four race bugs detected by Securify or Sailfish because it attempts to infer the conditions to trigger race bugs, whereas Securify and Sailfish only check the statements that are susceptible to race bugs. However, this also causes Securify to report 11 false positives and Sailfish to report 25 false positives. For instance, Securify suggests that a race bug can occur at Line 5 of Figure 5. However, since the contract does not have any function that can write to the variable *owner*, race access to *owner* can never occur.

Efficiency. As listed in Table 2, for the 50 smart contracts, TransRacer requires an average of 5.1 minutes to analyze a smart contract. The maximum, minimum, and median execution time of TransRacer is 21.8 minutes, 0.2 minutes, and 3.5 minutes, respectively. These values are reasonable because smart contracts that call more functions usually require checking more candidate function pairs. Given the 5.1-minute average, the average time required to perform static filtering and race checking at the initial contract states is 1.0 minutes and 0.9 minutes, respectively. For the 41 contracts involving function dependencies, TransRacer finds a total of 130 function dependencies with an average time cost of 1.8 minutes. The number of detected dependencies is 130, which led to race checking an average of 3.2 more states per contract. Consequently, TransRacer race detection averages 1.1 minutes and 1.7 minutes per contract at the initial and updated contract states, respectively. Therefore, TransRacer is efficient for multistate detection. In contrast, ETHRacer requires 31.1 minutes to analyze one smart contract on average, indicating that TransRacer is more efficient than ETHRacer. For the evaluated smart contracts, the execution time of the concrete executions is usually less than one second per smart contract. We do not discuss the time cost of the concrete executions as it has little impact on the overall time cost. Because Oyente, Securify, and Sailfish are lightweight and do not yield witness transactions, these three approaches require much less time than TransRacer.

We look into the smart contracts on which TransRacer takes more than ten minutes to finish the analysis, and find that these contracts usually involve many functions, functions dependencies, loops or branches, and race constraints that are not easy to solve.

Answer to RQ1: TransRacer detects 426 races, including 149 race bugs, in the 50 smart contracts without generating false positives. The number (percentages) of races and race bugs flagged at updated states are 63.1% (269/426) and 35.0% (52/149), respectively.
Implications: TransRacer can accurately find races (including race bugs) in smart contracts, and dependence analysis is helpful in race detection.

Table 3: The Numbers of Races and Race Bugs Detected in 6,943 Smart Contracts

	Initial state	Updated state	Total
$\#TR_I$	8,791	3,706	12,497
$\#TR_D$	4,479	7,490	11,969
Total	12,081	12,385	24,466
$\#TRB_I$	5,004	225	5,229
$\#TRB_D$	730	1,775	2,505
Total	5,734	2,000	7,734

Table 4: The Number of Smart Contracts Involving Races and Race Bugs

	Initial state	Updated state	Total unique
$\#TR_I$	3,535	1,661	3,664
$\#TR_D$	1,340	1,681	2,304
Total unique	3,539	1,861	3,680
$\#TRB_I$	2,589	143	2,620
$\#TRB_D$	451	871	1,111
Total unique	2,648	953	2,710

Answer to RQ2: For the 50 smart contracts, TransRacer spends an average of 5.1 minutes analyzing each smart contract. For the 41 smart contracts involving function dependencies, the average race detection times at the initial and updated contract states are 1.1 minutes and 1.7 minutes, respectively.
Implications: As a tool based on symbolic execution, TransRacer is efficient and scales well for race detection.

5.2 Empirical Study

To investigate the severity of races in practice, we further apply TransRacer to 6,943 real-world smart contracts. These smart contracts are initially used by ETHRacer to study event-order bugs [16]. We do not differentiate malignant and benign races. Tables 3 and 4 report the results of TransRacer on this large dataset.

Answer to RQ3: Of 6,943 smart contracts, TransRacer finds 3,680 (3,680/6,943 = 53.0%) smart contracts involving races. Of these, 2,710 (2,710/3,680 = 73.6%) lead to race bugs. Overall, TransRacer detects 24,466 races and 8,360 (7,734/24,466 = 31.6%) race bugs.
Implications: Races and race bugs are prevalent in real-world smart contracts.

For RQ3, the average numbers of races and race bugs among the 3,680 smart contracts are 6.6 and 2.1, respectively. These results indicate that races have been overlooked by developers.

The answer to RQ4 indicates that stateful exploration is helpful for race detection. For practitioners attempting to design a race detector, it is desirable to balance detection ability and efficiency.

Answer to RQ4: Among the 3,680 smart contracts that involve races, 1,861 (1,861/3,680 = 50.6%) and 953 (953/3,680 = 25.9%) lead to races and race bugs that can only be triggered at updated contract states, respectively.
Implications: Numerous races and race bugs are hidden at specific contract states. Therefore, in addition to the initial state, it is desirable to explore other reachable contract states to seek races.

Answer to RQ5: Among the 3,680 smart contracts, TransRacer finds 12,497 (12,497/24,466 = 51.1%) TR_I races in 3,664 (3,664/3,680 = 99.5%) smart contracts. Furthermore, 11,969 (11,969/24,466 = 48.9%) TR_D races are found in 2,304 (2,304/3,680 = 62.6%) smart contracts.
Implications: TR_I and TR_D races are prevalent in practice.

It is reasonable that TR_I races are common in smart contracts because two transactions that invoke the same function usually race to write the same variables. However, it is clear that there are also numerous instances of TR_D races. To prevent TR_D from occurring, a smart contract needs to precisely block unwanted transaction sequences and allow only the intended transaction sequences. Thus, the contract logic needs to be designed carefully.

Answer to RQ6: Races can cause consequences such as ether losses and token losses. For the 2,710 smart contracts that lead to race bugs, we find that 338 (338/2,710 = 12.5%) and 548 (548/2,710 = 20.2%) lead to bugs that result in ether and token losses, respectively.
Implications: Many real-world smart contracts involve race bugs that can lead to harmful consequences.

We verify whether a race bug that is relevant to two transactions can cause ether losses by checking whether executing two transactions in different orders leads to different contract balances. We verify the race bugs that cause token losses by checking whether at least one of the transactions can invoke the token transfer functions specified by the Ethereum token standards (e.g., ERC20, ERC721 [4]).

Figure 6 shows a money loss case found by TransRacer. One important task of this contract is to recruit money from an investor. The function `withdraw()` is used to withdraw the investment of the current investor. Suppose a new investor decides to join the project and the previous investor wants to quit and get a refund. Thus, the contract owner might invoke `setInvestor()` immediately after invoking `withdraw()`. If the miner executes `setInvestor()` before `withdraw()`, the previous investor will lose some money because his/her investment is withdrawn to the new investor. The intention of the contract designer is only to allow the investor to withdraw his/her own money. Since the contract behaviour is different from the intention of the contract designer, this race is malignant. The reason for this race is that both `withdraw()` and `setInvestor()` are callable at the contract state before the investor receives his/her investment. To prevent this race, one can add constraints to ensure that only `withdraw()` is callable at this contract state.

```
contract Preallocation{
   address investor;
   function withdraw() onlyOwner{
      uint bal = this.balance;
      if (!investor.send(bal)) {
         throw;}}
   function setInvestor(address _investor) onlyOwner
       {
      investor = _investor;}
   }
```

Figure 6: A transaction race leads to a money loss.

Answer to RQ7: The numbers (percentages) of TR_I and TR_D that can result in race bugs are 5,229 (5,229/12,497 = 41.8%) and 2,505 (2,505/12,497 = 20.0%), respectively. Moreover, we find that 1.7% (86/5,229) of the TRB_I and 47.8% (1,198/2,505) of the TRB_D can cause financial losses.
Implications: A TR_I is more likely to lead to race bugs than a TR_D. However, the race bugs induced by TR_D are more likely to cause financial losses.

The answer to RQ7 indicates that developers should pay attention to race bugs because many can lead to financial losses. Financial-loss cases usually emerge when one transaction affects the arguments (including *receiver*, *spender*, *amount*) of an ether (token) transfer operation initiated by other transactions. A TRB_D is more likely to cause money losses than a TRB_I because developers usually implement the token transfer configuration and token transfer initiation in two separate functions.

5.3 Threats to Validity and Limitations

Internal validity. TransRacer begins race checking at the initial state of each smart contract, because our approach is expected to be used by the developers before they deploy the contracts to the Ethereum. Our experimental results may vary if the state to begin the analysis is changed. In Section 5.2, we determine whether a race bug between two transactions can lead to token loss by checking whether at least one of the two transactions invokes a token transfer function. One may argue that if such transactions calling the token transfer function do not affect user balances, our results may be inaccurate in practice. Fortunately, we find that most of the smart contracts compliant with the token standards of Ethereum implement the transfer function solely for transferring tokens, and thus such race bugs can probably cause token losses.

Limitations. TransRacer uses function dependencies to seek reachable contract states that activate certain functions. However, this heuristic rule may not always work. Currently, we only consider transaction races in one smart contract, while inter-contract races are not considered.

6 RELATED WORK

Inconsistent contract state detection. A TOD bug is a race bug that leads to a difference in the contract balance. TOD bug detection techniques include symbolic execution, static analysis, and contract

fuzzing [1, 15, 21, 22, 34–37]. For example, Luu et al. [22] implement the first tool Oyente for detecting TOD bugs. It symbolically executes all the paths in the control flow graph of the smart contract to determine whether there are two paths that have different ether flows. ZEUS [15] first taints the global variables that are written to. Then, it determines whether there is a TOD bug by checking whether the tainted variables can influence the ether flow. Oyente and ZEUS employ symbolic execution only for path exploration, whereas TransRacer uses symbolic execution to capture constraints that directly lead to races. Securify [35] is a static analysis tool that detects TOD bugs by defining and checking the TOD violation patterns in the smart contract dependence graph. Sailfish [2] utilizes a light-weight exploration phase and a symbolic execution-based refinement phase to improve the efficiency and precision of the race detection. Several other TOD bug detectors [21, 37] use machine learning techniques to detect race bugs, but they do not generate witness transactions. ConFuzzius [34] tries to generate transactions that can trigger TOD bugs via fuzzing, while TransRacer generates witness transaction sequences with symbolic execution.

In addition to detecting ether flow differences, ETHRacer [16] is the first tool to detect differences at all variable states caused by the nondeterministic transaction execution order. The authors first employ symbolic execution to generate candidate transactions for fuzzing and then check the fuzzing results to determine whether different execution sequences result in different contract states. In contrast to ETHRacer, TransRacer aims to detect transaction (data) races between two function invocations. Moreover, instead of exploring different contract states through fuzzing, TransRacer infers function dependencies to seek contract states that can activate certain functions. As a result, TransRacer finds more smart contracts that involve race bugs than ETHRacer.

Dependence-based bug detection. Different notions of dependence have been proposed to help detect smart contract bugs [9, 31, 34, 40, 42, 43]. The framework for Ethereum transaction attack detection (TXSpector) [40] first determines whether an opcode `SLOAD` (`JUMPI`) depends on another opcode `SSTORE` to capture the data(control) flow dependence in a program path. Then, it uses these dependencies to construct rules for bug detection, such as reentrancy bugs and unchecked calls. Slither [9] is a static analysis tool that models data dependencies and uses them as features to trigger bugs such as uninitialized variables and reentrancy bugs. Symbolic value-flow static analysis (symvalic) [31] is a static bug detector that employs symbolic analysis to capture the dependencies among a small set of values to improve the precision and scalability of the detection results. MPro [41] utilizes the data dependence between two functions to reduce the number of paths that need to be symbolically executed. Torres et al. [34] propose a genetic algorithm that leverages data dependence analysis to rearrange the transaction order and generate new populations for fuzzing. In contrast to these studies, we define and use function dependencies to help detect contract bugs and yield witness transaction sequences.

Transaction sequence-based bug detection. Many detectors aim to generate transaction sequences to detect hidden smart contract bugs. Most of them are based on techniques of symbolic execution [11, 17, 24, 26, 41], fuzzing [14, 19, 25, 38, 39], machine learning [13, 32], and model checking [33]. For instance, Manticore [24] and Teether [17] detect different bugs by symbolically executing a sequence of functions and checking whether the safety violations exist in the executed paths. Fuzzing-based tools such as ConFuzzer [14] and ReGuard [19] randomly generate transaction sequences to trigger smart contract bugs. Harvey [38] and sFuzz [25] generate transaction sequences by randomly mutating the execution orders of their selected seed transactions. Similarly, Fluffy [39] utilizes a multi-transaction differential fuzzer to manifest consensus bugs in Ethereum. Imitation learning-based fuzzer (ILF) [13] and Smartest [32] adopt machine learning techniques to learn a predictive model [27, 29] to generate transaction sequences to trigger bugs. Smartian [6] employs static analysis to generate transaction sequences as seeds. Then, it leverages data flow analysis to mutate these seeds for fuzzing. Notably, the detectors based on symbolic execution win in accuracy, while those based on fuzzing and machine learning win in efficiency. To balance accuracy and efficiency, TransRacer employs function dependencies and concrete execution to guide the procedure of transaction race detection with symbolic execution. To generate transaction sequences that violate the liveness properties of smart contracts, SmartPulse [33] initially invokes a random sequence of functions with random values to generate transaction sequences. Then, it checks whether there is a sequence that violates the liveness properties.

Existing race detection techniques for traditional programming languages such as C/C++ or Java mostly focus on detecting low-level data races, caused by missing locks or improper synchronizations between parallel threads (see classical papers [10] and [18]). However, in smart contracts, the races are high-level logical races due to abnormal or non-deterministic transaction orderings. Existing techniques cannot be easily adapted to detect transaction races because all transactions are executed by a single thread (miner) sequentially and locks are not available.

7 CONCLUSIONS

In this paper, we present TransRacer, a tool that combines symbolic execution and concrete execution for transaction race detection in Ethereum smart contracts. By leveraging symbolic execution and constraint solving, TransRacer effectively generates transactions that can trigger races. TransRacer employs function dependence analysis and concrete execution to find contract states that can activate certain functions, which allows TransRacer to find more races hidden at specific contract states. An experimental evaluation of 50 real-world smart contracts demonstrates the effectiveness and efficiency of TransRacer. An empirical study on 6,943 smart contracts demonstrates that races are prevalent in practice, and many race bugs can cause ether or token losses.

DATA AVAILABILITY

All artifacts, including TransRacer[3] and the datasets (smart contract addresses) of the article are publicly available [23].

ACKNOWLEDGMENTS

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61761136003.

[3] https://github.com/wsong-nj/TransRacer

REFERENCES

[1] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis, ATVA'18, Los Angeles, CA, USA, October 7-10 (Lecture Notes in Computer Science, Vol. 11138)*. Springer, Los Angeles, CA, USA, 513–520. https://doi.org/10.1007/978-3-030-01090-4_30

[2] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds. In *Proceedings of the 43rd IEEE Symposium on Security and Privacy, SP'22, San Francisco, CA, USA, May 22-26*. IEEE, San Francisco, CA, USA, 161–178. https://doi.org/10.1109/SP46214.2022.9833721

[3] Vitalik Buterin et al. 2014. A next-generation smart contract and decentralized application platform. *white paper* 3, 37 (2014), 2–1.

[4] Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. 2020. Native Custom Tokens in the Extended UTXO Model. In *Proceedings of the 9th International Symposium on Leveraging Applications of Formal Methods Part III - Leveraging Applications of Formal Methods, Verification and Validation: Applications, ISoLA'2020, Rhodes, Greece, October 20-30*, Vol. 12478. Springer, Rhodes, Greece, 89–111. https://doi.org/10.1007/978-3-030-61467-6_7

[5] Weili Chen, Tuo Zhang, Zhiguang Chen, Zibin Zheng, and Yutong Lu. 2020. Traveling the token world: A graph analysis of Ethereum ERC20 token ecosystem. In *Proceedings of the The Web Conference, WWW'20, Taipei, Taiwan, April 20-24*. ACM/IW3C2, Taipei, Taiwan, China, 1411–1421. https://doi.org/10.1145/3366423.3380215

[6] Jaeseung Choi, Doyeon Kim, Soomin Kim, Gustavo Grieco, Alex Groce, and Sang Kil Cha. 2021. SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE'21, Melbourne, Australia, November 15-19*. IEEE, Melbourne, Australia, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888

[7] Chris Dannen. 2017. *Introducing Ethereum and solidity* (1 ed.). Springer. 1–185 pages. https://doi.org/10.1007/978-1-4842-2535-6

[8] Thomas Durieux, João F. Ferreira, Rui Abreu, and Pedro Cruz. 2020. Empirical review of automated analysis tools on 47, 587 Ethereum smart contracts. In *Proceedings of 42nd International Conference on Software Engineering ICSE'20, Seoul, South Korea, 27 June - 19 July*. ACM, 530–541. https://doi.org/10.1145/3377811.3380364

[9] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE'19, Montreal, QC, Canada, May 27*. IEEE/ACM, Montreal, QC, Canada, 8–15. https://doi.org/10.1109/WETSEB.2019.00008

[10] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'09, Dublin, Ireland, June 15-21*. ACM, 121–133. https://doi.org/10.1145/1542476.1542490

[11] Joel Frank, Cornelius Aschermann, and Thorsten Holz. 2020. ETHBMC: A Bounded Model Checker for Smart Contracts. In *Proceedings of the 29th USENIX Security Symposium, USENIX Security'20, August 12-14*. USENIX Association, 2757–2774. https://www.usenix.org/conference/usenixsecurity20/presentation/frank

[12] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. 2020. What are the Actual Flaws in Important Smart Contracts (And How Can We Find Them)?. In *Proceedings of the Financial Cryptography and Data Security - 24th International Conference, FC'20, Kota Kinabalu, Malaysia, February 10-14*. Springer, Kota Kinabalu, Malaysia, 634–653. https://doi.org/10.1007/978-3-030-51280-4_34

[13] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS'19, London, UK, November 11-15*. ACM, London, UK, 531–548. https://doi.org/10.1145/3319535.3363230

[14] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE'18, Montpellier, France, September 3-7*. ACM, Montpellier, France, 259–269. https://doi.org/10.1145/3238147.3238177

[15] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: Analyzing Safety of Smart Contracts. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium, NDSS'18, San Diego, California, USA, February 18-21*. The Internet Society, San Diego, California, USA, 1–15. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[16] Aashish Kolluri, Ivica Nikolic, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2019. Exploiting the laws of order in smart contracts. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'19, Beijing, China, July 15-19*. ACM, Beijing, China, 363–373. https://doi.org/10.1145/3293882.3330560

[17] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *Proceedings of the 27th USENIX Security Symposium, USENIX Security'18, Baltimore, MD, USA, August 15-17*. USENIX Association, Baltimore, MD, USA, 1317–1333. https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[18] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. 2021. When threads meet events: efficient and precise static race detection with origins. In *PLDI'21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25*. ACM, 725–739. https://doi.org/10.1145/3453483.3454073

[19] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. ReGuard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE'18, Gothenburg, Sweden, May 27 - June 03*. ACM, Gothenburg, Sweden, 65–68. https://doi.org/10.1145/3183440.3183495

[20] Lu Liu, Lili Wei, Wuqi Zhang, Ming Wen, Yepang Liu, and Shing-Chi Cheung. 2021. Characterizing Transaction-Reverting Statements in Ethereum Smart Contracts. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE'2021, Melbourne, Australia, November 15-19*. IEEE, Melbourne, Australia, 630–641. https://doi.org/10.1109/ASE51524.2021.9678597

[21] Oliver Lutz, Huili Chen, Hossein Fereidooni, Christoph Sendner, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Farinaz Koushanfar. 2021. ESCORT: Ethereum Smart COntRacTs Vulnerability Detection using Deep Neural Network and Transfer Learning. *CoRR* abs/2103.12607 (2021), 1–17. https://arxiv.org/abs/2103.12607

[22] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS'16, Vienna, Austria, October 24-28*. ACM, Vienna, Austria, 254–269. https://doi.org/10.1145/2976749.2978309

[23] Chenyang Ma, Wei Song, and Jeff Huang. 2023. Reproduction Package of 'TransRacer: Function Dependence-Guided Transaction Race Detection for Smart Contracts'. https://doi.org/10.5281/zenodo.8198972

[24] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A User-Friendly Symbolic Execution Framework for Binaries and Smart Contracts. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering, ASE'19, San Diego, CA, USA, November 11-15*. IEEE, San Diego, CA, USA, 1186–1189. https://doi.org/10.1109/ASE.2019.00133

[25] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the 42nd International Conference on Software Engineering, ICSE'20, Seoul, South Korea, 27 June - 19 July*. ACM, Seoul, South Korea, 778–788. https://doi.org/10.1145/3377811.3380334

[26] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC'18, San Juan, PR, USA, December 03-07*. ACM, San Juan, PR, USA, 653–663. https://doi.org/10.1145/3274694.3274743

[27] Veselin Raychev, Martin T. Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, Edinburgh, United Kingdom - June 09 - 11*. ACM, Edinburgh, United Kingdom, 419–428. https://doi.org/10.1145/2594291.2594321

[28] Meng Ren, Zijing Yin, Fuchen Ma, Zhenyang Xu, Yu Jiang, Chengnian Sun, Huizhong Li, and Yan Cai. 2021. Empirical evaluation of smart contract testing: what is the best choice?. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '21, Virtual Event, Denmark, July 11-17*. ACM, 566–579. https://doi.org/10.1145/3460319.3464837

[29] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics, AISTATS'11, Fort Lauderdale, USA, April 11-13*, Vol. 15. JMLR.org, Fort Lauderdale, USA, 627–635. http://proceedings.mlr.press/v15/ross11a/ross11a.pdf

[30] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *Proceedings of the Financial Cryptography and Data Security - International Workshops, FC'17, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, Revised Selected Papers*, Vol. 10323. Springer, Sliema, Malta, 478–493. https://doi.org/10.1007/978-3-319-70278-0_30

[31] Yannis Smaragdakis, Neville Grech, Sifis Lagouvardos, Konstantinos Triantafyllou, and Ilias Tsatiris. 2021. Symbolic value-flow static analysis: deep, precise, complete modeling of Ethereum smart contracts. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. https://doi.org/10.1145/3485540

[32] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution. In *Proceedings of the 30th USENIX Security Symposium, USENIX Security'21, August 11-13*. USENIX Association, 1361–1378. https://www.usenix.org/conference/usenixsecurity21/presentation/so

[33] Jon Stephens, Kostas Ferles, Benjamin Mariano, Shuvendu K. Lahiri, and Isil Dillig. 2021. SmartPulse: Automated Checking of Temporal Properties in Smart Contracts. In *42nd IEEE Symposium on Security and Privacy, SP'2021, San Francisco, CA, USA, 24-27 May*. IEEE, 555–571. https://doi.org/10.1109/SP40001.2021.00085

[34] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *Proceedings of the IEEE European Symposium on Security and Privacy, EuroS&P'21, Vienna, Austria, September 6-10*. IEEE, Vienna, Austria, 103–119. https://doi.org/10.1109/EuroSP51992.2021.00018

[35] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS'18, Toronto, ON, Canada, October 15-19*. ACM, Toronto, ON, Canada, 67–82. https://doi.org/10.1145/3243734.3243780

[36] Shuai Wang, Chengyu Zhang, and Zhendong Su. 2019. Detecting nondeterministic payment bugs in Ethereum smart contracts. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 189:1–189:29. https://doi.org/10.1145/3360615

[37] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2021. ContractWard: Automated Vulnerability Detection Models for Ethereum Smart Contracts. *IEEE Trans. Netw. Sci. Eng.* 8, 2 (2021), 1133–1144. https://doi.org/10.1109/TNSE.2020.2968505

[38] Valentin Wüstholz and Maria Christakis. 2020. Harvey: a greybox fuzzer for smart contracts. In *Proceedings of the ACM 28th Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE'20, Virtual Event, USA, November 8-13*. ACM, USA, 1398–1409. https://doi.org/10.1145/3368089.3417064

[39] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. 2021. Finding Consensus Bugs in Ethereum via Multi-transaction Differential Fuzzing. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI'21, July 14-16*. USENIX Association, 349–365. https://www.usenix.org/conference/osdi21/presentation/yang

[40] Mengya Zhang, Xiaokuan Zhang, Yinqian Zhang, and Zhiqiang Lin. 2020. TXSPECTOR: Uncovering Attacks in Ethereum from Transactions. In *Proceedings of the 29th USENIX Security Symposium, USENIX Security'20, August 12-14*. USENIX Association, 2775–2792. https://www.usenix.org/conference/usenixsecurity20/presentation/zhang-mengya

[41] William Zhang, Sebastian Banescu, Leonardo Pasos, Steven T. Stewart, and Vijay Ganesh. 2019. MPro: Combining Static and Symbolic Analysis for Scalable Testing of Smart Contract. In *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering, ISSRE'19, Berlin, Germany, October 28-31*. IEEE, Berlin, Germany, 456–462. https://doi.org/10.1109/ISSRE.2019.00052

[42] Yuyao Zhang, Siqi Ma, Juanru Li, Kailai Li, Surya Nepal, and Dawu Gu. 2020. SMARTSHIELD: Automatic Smart Contract Protection Made Easy. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER'2020, London, ON, Canada, February 18-21*. IEEE, London, ON, Canada, 23–34. https://doi.org/10.1109/SANER48275.2020.9054825

[43] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2020. Smart Contract Vulnerability Detection using Graph Neural Network. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI'20*. ijcai.org, 3283–3290. https://doi.org/10.24963/ijcai.2020/454

Received 2023-02-02; accepted 2023-07-27