Exploiting the Laws of Order in Smart Contracts

Aashish Kolluri
School of Computing, NUS
Singapore

Ivica Nikolic
School of Computing, NUS
Singapore

Ilya Sergey
Yale-NUS College
School of Computing, NUS
Singapore

Aquinas Hobor
Yale-NUS College
School of Computing, NUS
Singapore

Prateek Saxena
School of Computing, NUS
Singapore

ABSTRACT

We investigate a family of bugs in blockchain-based smart contracts, which we dub *event-ordering* (or EO) bugs. These bugs are intimately related to the dynamic ordering of contract *events*, *i.e.* calls of its functions, and enable potential exploits of millions of USD worth of crypto-coins. Previous techniques to detect EO bugs have been restricted to those bugs that involve just one or two event orderings. Our work provides a new formulation of the general class of EO bugs arising in *long* permutations of such events by using techniques from concurrent program analysis.

The technical challenge in detecting EO bugs in blockchain contracts is the inherent combinatorial blowup in path and state space analysis, even for simple contracts. We propose the first use of partial-order reduction techniques, using automatically extracted happens-before relations along with several dynamic symbolic execution optimizations. We build EthRacer, an automatic analysis tool that runs directly on Ethereum bytecode and requires no hints from users. It flags 8% of over 10,000 contracts analyzed, providing compact event traces (witnesses) that human analysts can examine in only a few minutes per contract. More than half of the flagged contracts are likely to have unintended behavior.

CCS CONCEPTS

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

KEYWORDS

Smart Contract Security, Concurrency, Ethereum, Happens-Before

ACM Reference Format:
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), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330560

ISSTA '19, July 15–19, 2019, Beijing, China

ACM ISBN 978-1-4503-6224-5/19/07…$15.00
https://doi.org/10.1145/3293882.3330560

1 INTRODUCTION

A blockchain/cryptocurrency protocol enables a distributed network of mutually-untrusting computational nodes (*miners*) to agree on the current state and complete history of a replicated public ledger. The dominant consensus algorithm was invented to facilitate decentralized payments in virtual currencies [37], but it has since been extended to the decentralized applications commonly known as *smart contracts* [48]. A typical smart contract on a blockchain is a stateful program, *i.e.*, a package of code and the mutable data that describes the contract's current state, similar to an object in an OOP language. Both the code and the data are stored in a replicated fashion on the blockchain. Smart contract transactions (invocations of contract code) are totally ordered, as agreed upon by a majority of miners, and replicated across the system.

Smart contracts implement domain-specific logic to act as automatic and trustworthy mediators. Typical applications include multi-party accounting, voting, auctions, and puzzle-solving games with a distribution of rewards. Numerous publicly reported attacks have resulted in hundreds of millions dollars' worth of Ether being stolen or otherwise lost [13, 30]. Further, contracts cannot be patched once deployed. This emphasizes the importance of pre-deployment security audits and analyses for smart contracts.

This paper investigates a class of vulnerabilities in smart contracts that arise due to their inherent *concurrent* execution model. Contracts can be invoked by multiple users concurrently, and the ordering of multiple submitted transactions is non-deterministically decided by miners through a consensus protocol. Contracts can invoke other contracts *synchronously* and call off-chain services *asynchronously* which return in no pre-determined order. As Ethereum contracts are *stateful*, mutations of contract data persist between invocations. Therefore, predicting the result from a set of transactions invoking a contract requires reasoning about the non-deterministic order of concurrently-interacting transactions. Developers often write contracts assuming a certain serialized execution order of contracts, often missing undesirable behaviors only observable in complex interleavings. Reasoning about interleaved executions has historically been difficult for human auditors. Accordingly, tools that allow developers, auditors, and smart contract users to increase confidence that contracts behave as expected are useful.

Certain concurrency bugs in Ethereum smart contracts are known. Prior work has highlighted the susceptibility of contracts to asynchronous callbacks [5, 43]; and how a pair of transactions, when reordered, can cause contracts to exhibit differing Ether transfers as

output [30]. However, the full generality of bugs arising from unexpected ordering of *events—i.e.* calls to contract functions invoked via transactions and callbacks—has neither been systematically tested nor fully understood to date. The key challenge is that analyzing contracts under multiple events spread over many transactions leads to *combinatorial blowup* in the analyzed state-space. Existing tools shrink the search space by checking for properties of a single event or a single pair of events. For instance, the infamous DAO reentrancy bug can be found by checking whether a function can call itself within a single transaction execution [21], while the Oyente tool checks a pair of events for transaction ordering bugs [30].

Problem & Approach. In this work, we develop new and efficient analysis techniques for Ethereum smart contracts under *multiple* events. Our work generalizes beyond several previous classes of errors into a broader category of concurrency errors we call *event-ordering* (EO) bugs. The core idea is to check whether changing the ordering of events (function invocations) of a contract results in *differing* outputs. If a contract exhibits differing outputs under reordered events, it is flagged as an EO bug; otherwise, event reordering has no impact on outputs and so the contract is EO-safe.

To tackle combinatorial path and state explosion, we develop a number of optimization techniques. Furthering the "contracts-as-concurrent-objects" analogy, we show that partial-order reduction techniques can be applied to contract analysis. Specifically, we can shrink the number of event combinations (traces) over a set of functions S that we must consider if we determine that the functions in S can be non-erroneously invoked in only certain orders. This concept is captured by the classical *happens-before* (HB) relation [29]. Unlike traditional programming languages with explicit synchronization primitives, smart contracts try to implement desirable concurrency controls using ad-hoc program logic and global state. We show how to recover the intrinsic HB-relation encoded in contract logic, and that it substantially reduces the event combinations we must check.

EthRacer. Our central practical contribution is EthRacer, an automatic tool to find EO bugs. We use EthRacer to measure the prevalence of EO vulnerabilities in over ten thousand contracts. Less than 1% of live contracts are accompanied by source code; hence, our tool does not require source and analyzes Ethereum bytecode directly. This enables third-party audit and testing of contracts without source. We take a dynamic testing approach, constructing inputs and systematically trying all possible function orderings until a budgeted timeout is reached. Done naïvely, this approach would quickly lead to an intractable analysis even for relatively small contracts, since N function calls to a contract can have $N!$ orderings. Our approach combines symbolic execution of contract events with fast randomized fuzzing of event sequences. Our key optimizations exponentially reduce the search space by eliminating orderings which violate the induced HB-relation between events.

EthRacer reports only true EO violations, accompanied by witnesses of event values that can be concretely executed.

Emperical Results. First, we show that most contracts *do not* exhibit differences in outputs. EthRacer flags 836 (8%) out of over 10,000 analyzed contracts. When contracts *do* exhibit different outputs upon re-ordering, we find that they are likely to have an unintended behavior more than half of the time. Therefore, our formulation of properties catches a subtle and dangerous class of EO bugs without excessively triggering alarms.

Second, EthRacer minimizes human analysis effort. When EthRacer reports EO violations, it provides concrete witnesses that exhibit differing outputs. Typically there are very few (*e.g.*, 1–3) witnesses that require human inspection, a process that requires only a few minutes per contract. Since contracts are not patchable after deployment, we believe that EthRacer is useful auditing tool.

Third, we show that our characterization of EO bugs substantially generalizes beyond bugs known from prior works. A direct comparison to prior work, Oyente, shows that EthRacer finds *all* 78 true EO bugs Oyente finds as well as many that are not detected by Oyente (674 in total). Also, EthRacer flags 47 contracts by analyzing asynchronous callbacks into these contracts, by off-chain services. Such analysis is not captured by any prior work, generalizing this class of errors beyond reentrancy [13] or id-tracking bugs that fail to pair asynchronous calls and callbacks [43].

2 MOTIVATION

2.1 Ethereum Smart Contracts

Smart contracts in Ethereum are identified by addresses. A user invokes a particular function of a smart contract by creating a signed transaction to the contract's address. The transaction specifies which function is being executed and its arguments. The user submits the transaction to the Ethereum network, and at some point in the future, a *miner* in the network chooses to process the transaction. To do so, the miner takes the current state of the contract from the blockchain, executes the invoked function, and stores the updated state of the contract back into the blockchain.

It is common for two users to submit transactions that interact with the same contract concurrently. Neither of the users knows whose transaction will be mined first. If a user's transaction is incorporated into the blockchain then its effect will be reflected atomically. In other words, a miner will *not* execute part of the first transaction, switch to running the second at some intermediate contract state, and then return to finish off the remaining computation in the first transaction. It is easy to assume that this *transaction atomicity* removes the need to reason about the concurrent execution environment, but as we explain next this is unsound.

2.2 Event-Ordering Bugs

Contracts can be seen as objects with a mutable state and a set of interfaces for users to access such state. These interfaces can be invoked by many users simultaneously; the order in which these invocations (transactions) will be executed is determined entirely by the miners. Moreover, contracts can access off-chain services for various purposes. The replies from these services can be asynchronous, leading to a nondeterministic ordering on the blockchain.

Off-chain asynchronous callbacks. Consider the `Casino` snippet in Figure 1, which was simplified from a real smart contract. `Casino` accepts bets from one or more players and, with 200-to-1 odds (Line 11), repays winners 100-fold (Line 12). `Casino` aims to be honest, so it rejects any bet it cannot honor (Line 4). The fairness of any game of chance depends on how random values are generated. `Casino` invokes a trusted off-chain random number generator using the Oraclize API [39] query (Line 5). The random-number oracle is *not* actually queried in Line 5. Calling `oraclize_query` generates the unique transaction id tag `oid` and notifies a trusted off-chain

```
1  contract Casino {
2    function bet() payable {
3      // make sure we can pay out the player
4      if (address(this).balance < msg.value * 100 ) throw;
5      bytes32 oid = oraclize_query(...); // random
6      bets[oid] = msg.value;
7      players[oid] = msg.sender;  }
8    function __callback(bytes32 myid, string result)
9      onlyOraclize onlyIfNotProcessed(myid) {
10     ...
11     if (parseInt(result) % 200 == 42)
12     players[myid].send( bets[myid] * 100 );  }...}
```

Figure 1: Contract `Casino` with an asynchronous callback.

monitor (the Oraclize service) that `Casino` wishes to query the random-number oracle for transaction `oid`. Due to the semantics of Ethereum, the off-chain monitor will be notified only *after* `Casino`'s code has finished running (Line 7). Later, once the off-chain oracle has been queried, the Oraclize service will make a fresh Ethereum transaction to invoke `Casino`'s callback function `__callback` (Line 8).

The `__callback` function "returns" asynchronously. After an initial bettor initiates an oracle query, other bettors can place their bets while the off-chain oracle is queried. These further wagers will initiate further oracle queries, and depending on the behavior of the off-chain oracles, their corresponding callbacks may not be invoked in the same order as they are called. The designers of the Oraclize API are aware of this, which is why each transaction is given a unique ID that is both returned from `oraclize_query` (Line 5) and passed to the callback (the `myid` parameter in Line 8), thereby "pairing" the two. Failing to pair callbacks can lead to previously-published vulnerabilities [43], but this error is avoided by `Casino`.

Even though the call/return are correctly paired, there is a bug that can occur when multiple players place bets concurrently. Suppose that the contract has 100 Ether and that two players wish to bet 1 Ether. Consider the following execution of functions: $\langle \texttt{bet}_1; \texttt{bet}_2; \texttt{\_\_callback}_1; \texttt{\_\_callback}_2 \rangle$, where the subscript denotes the paired identifier (`oid`/`myId`) in the Oraclize interface. Both bets are accepted (Line 4), but if both bets win (Line 11), player two will not be paid anything. A fairer `Casino` implementation should consider all pending bets when determining whether to accept another bet.

This contract yields differing outputs if the callbacks are received out-of-order. The ordering presented above yields an insufficient balance after paying off player 1 (Line 12). In the alternative ordering $\langle \texttt{bet}_1; \texttt{\_\_callback}_1; \texttt{bet}_2; \texttt{\_\_callback}_2 \rangle$, $\texttt{bet}_2$ will decline the second bet due to the check on Line 4, saving the second bettor's money.

On-chain transaction ordering. Smart contracts implicitly desire to order multiple user requests in a particular sequence, but sometimes fail to. As an example, Figure 2 shows a shortened version of a contract that implements ERC-20-compliant tokens in Ethereum dubbed "IOU"s. As of today, ERC-20-compliant contracts manage tokens valued in the billions of USD. The snippet in Figure 2 allows an owner "O" of IOUs to delegate control of a certain `_val` of IOU tokens to a specified `_spender` "S'' (*e.g.*, an expense account). O calls the `approve` function (Line 3) to allocate `_val` IOU tokens to S, and the function transferFrom allows S to send a portion (`_val`) of the IOU allocation to an address of S's choice (`_to`). The approver can update the allocation any time: for instance, O may initially approve 300 IOU and later reduce the amount to 100 by calling

```
1  contract IOU {
2   // Approves the transfer of tokens
3   function approve(address _spender, uint256 _val) {
4     allowed[msg.sender][_spender] = _val;
5     return true; }
6   // Transfers tokens
7   function transferFrom(address _from, address _to,
8                         uint256 _val) {
9     require(allowed[_from][msg.sender] >= _val
10            && balances[_from] >= _val
11            && _val > 0);
12    balances[_from] -= _val;
13    balances[_to] += _val;
14    allowed [_from][msg.sender] -= _val;
15    return true; }...}
```

Figure 2: Contract `IOU` with on-chain transaction ordering

`approve` again. Although this may seem like a reasonable idea, as pointed out on public forums [1], this contract has undesirable semantics, since S can execute a `transferFrom` between the two calls to `approve`, thereby spending the first allocation of 100 and then having another 100 to spend. Here, two different executions of calls to `approve` and `transferFrom` lead to different outcomes:

Execution 1	Execution 2
$\texttt{approve}_O\texttt{(S, 300)}$	$\texttt{approve}_O\texttt{(S, 300)}$
$\texttt{approve}_O\texttt{(S, 100)}$	$\texttt{transferFrom}_S\texttt{(O, S, 100)}$
$\texttt{transferFrom}_S\texttt{(O, S, 100)}$	$\texttt{approve}_O\texttt{(S, 100)}$
S has spent 100 and can spend 0 more	*S has spent 100 and can spend 100 more*

The community has proposed a fix, forbidding a change an allocation once made [1]. The recommendation ensures that all calls to `transferFrom` by all users should be permitted strictly after all `approve` have happened. Newer ERC-20 contracts have deployed this fix and behave more reasonably if such order is enforced, since the second `approve` is rejected in the second ordering.

3 OVERVIEW

We discuss a class of bugs which we call *event-ordering bugs* (formally in Section 4.1), generalizing the examples presented in Section 2. We explain the inherent path and state space exploration complexity and propose our design to address these systematically.

3.1 Problem

A contract function can be invoked by an external transaction, an internal call from another contract, or via an off-chain asynchronous callback. We call these invocations *events*. Under a received event, a contract executes atomically, which we call a *run*. A sequence of events invokes a sequence of contract runs, which we call a *trace*. Each run of a contract can modify its Ether balance and global state as well as generate new transactions that transfer Ether or call other contracts. We say that the *output* of a run or a trace are values of the contract balance, state, and any resulting transactions. We define these terms more precisely in Section 4.1. An event-ordering (EO) bug exists in a contract if two different traces, consisting of the same set of concrete events, produce different outputs. We seek to check if a given smart contract has an event-ordering (EO) bug.

Our problem formulation is more general than previous work, *e.g.* transaction ordering bugs (TOD) [30], which only deal with a

pair of events that change a contract's balance. Also, our EO bug formulation is orthogonal to the concept of reentrancy bugs, *e.g.* the DAO, which do not correspond to re-ordering of *multiple* events. Re-entrancy bugs are already detected by several tools that analyze runs from different values for a *single* event [21, 30].

Intent vs. Bugs. Unlike dereferencing a null pointer, which is widely-understood as a bug, the bugs uncovered in the present work tend to arise from logical errors between a contract's intended behavior and its implementation. Unfortunately, smart contracts are usually deployed anonymously and without public documentation; the true motivations for their deployment can be very obscure.

Therefore, our focus in this work is merely on efficient analysis techniques that minimize the number of test configurations the developer or potential user has to manually inspect, before the contract is deployed irrevocably.

3.2 Our Solution

We take a dynamic testing approach to find EO bugs. This is primarily motivated by our goal to produce concrete witnesses that human analysts can replay to inspect and confirm EO bugs with no false violations. The technical barrier to finding reordering bugs is that the path and the state space of the analysis blows up combinatorially. To illustrate this, consider a baseline solution that (say) can reason about the behavior of the contract under different values of a single input event. One effective state-of-the-art technique for such analysis is based on dynamic symbolic execution (DSE), implemented by many existing analyses for contracts [26, 27, 38].

In concept, such a DSE engine can enumerate different paths and input values (symbolically) in the program, starting from one event entry point function, and check if two different paths could lead to different outputs. However, this baseline solution does *not* address the class of EO bugs sought directly. EO bugs are a result of two or more events with possibly different entry points, leading to two different paths and outputs. Therefore, even with a powerful DSE engine, checking the space of N events is prohibitive. For instance, even a (relatively short) sequence of calls to a contract with (say) 20 functions callable via events can have millions of traces to inspect[1]. Therefore, we seek techniques that can eliminate a large part of this path and state space exploration, making search tractable.

Key ideas. The first observation is that contract functions are often written in a way that have a certain intended order of *valid* execution. Executing them in a different order simply results in the contract throwing an exception. This can be captured as a partial-order between events or as a happens-before (or *hb*) relation. Two events e_1 and e_2 are in a *hb* relation if executing them in one order produces a valid trace while the other is invalid. The insight is that if we find pairs of events e_1 and e_2 that are in $hb(e_1, e_2)$ relation, *all*[2] permutations of events involving these in an invalid order will result in exception—therefore, these are redundant to test repeatedly. To maximize the benefit of this observation, we augment the baseline DSE to recognize events which are in the *hb* relation. The symbolic analysis reasons about a large set of event values encoded as symbolic path constraints, which helps to identify partially-ordered events much better than fuzzing with concrete values. We present our novel and efficient algorithm to extract a new notion of *hb* relations we define in Section 4.3.

Our second observation is that certain events produce the same outputs irrespective of which order they appear in any trace. A simple, but highly effective, optimization is to recognize events runs that do not write or read global state. Such events can be arbitrarily reordered without changing outputs, so testing permutations that simply reorder these is unnecessary. Similarly, events that do not write to any shared global state can be arbitrarily reordered. Hence, they are also not considered for inducing an HB relation.

The final key observation deals primarily with optimizing for contracts with asynchronous callbacks. If we consider traces where each off-chain request and its matching callbacks are sequentially or atomically executed, these orders likely yield intended (benign) outputs. Thus, we consider such traces in which events corresponding to different user requests are not "out-of-order" to be linearizable traces [22]. Now, the task of analysis reduces to finding traces that do *not* produce outputs same as that of some linearizable trace.

4 ETHRACER DESIGN

The notion of concurrency in contracts requires a careful mapping to the familiar concepts of concurrency. We begin by recalling the contract concurrency model and by precisely defining terminology used informally thus far. We utilize the notation used in a distilled form of Ethereum Virtual Machine (EVM) semantics called EtherLite calculus [30]. Then, we move to EthRacer's design.

4.1 Contract Concurrency Model

Recall that each event executes atomically and deterministically in Ethereum [50]. A miner runs the contract function, conceptually as a single thread, with provided inputs of an event until it either terminates or throws an exception.[3] If it terminates successfully, all the changes to the global variables of that execution are committed to the blockchain; if an exception is thrown, none of the changes during the execution under that event are committed to the blockchain. In this sense, the execution of one event is atomic. The source of concurrency lies in the non-deterministic choices that each miner makes in ordering transaction in a proposed block.

Contract states and instances. We recall the definitions of global blockchain state, contracts and messages from EtherLite calculus.

A global blockchain state σ is encoded as a finite partial mapping from an account *id* to its balance, contract code M and its *mutable* state—which is a mapping from field names fld to the corresponding values. Contract code M and its *mutable* state , are optional (marked with "?") and are only present for contract-storing blockchain records. We refer to the union of a contract's field entries $\overline{\mathsf{fld} \mapsto v}$ and its balance entry $\mathsf{bal} \mapsto z$ as a *contract state* ρ,[4] and denote a triple $c = \langle id, M, \rho \rangle$ of a contract with an account *id*, the code of which is M and state is ρ, as *contract instance* c.

[1] *E.g.*, for a typical contract with 20 functions, each with 2 different inputs, the number of traces composed of 6 events is $C^6_{(20 \cdot 2 \cdot 6)} \approx 2^{38}$.

[2] The savings can be exponential, since there are exponentially many permutations with a shared pair of reordered events.

[3] Here, we do not draw a difference between different origins of exceptions, *i.e.*, those raised programmatically or those triggered by insufficient *gas*.

[4] We will also overload the notation, referring to a state ρ of a contract *id* in a blockchain state σ as $\rho = \sigma[id]$, thus, ignoring its code component.

$$\sigma \triangleq \overline{id \mapsto \left\{\text{bal} : \mathbb{N};\ \text{code?} \mapsto M;\ \overline{\text{fld?} \mapsto v}\right\}}$$

Messages (ranged over by m) are encoded as mappings from identifiers to heterogeneous values. Each message stores the identity of its sender and destination (to), the amount value of Ether being transferred (represented as a natural number), a contract function name to invoke, and auxiliary fields (data) containing additional arguments for a contract function, which we omit for brevity.

$$m \triangleq \{\text{sender} \mapsto id; \text{to} \mapsto id';\ \text{value} : \mathbb{N};\ \text{fname} : string\ ; \text{data} \mapsto \ldots\}$$

We will refer to a value of an entry x of a message m as $m.x$.

Contract events. The notion of events captures external inputs that can force control into an entry point of a contract, defined as:

Definition 1 (Events). *An* event *e for a contract instance* $c = \langle id, M, \rho\rangle$ *is a pair* $\langle fn, m\rangle$*, where fn is a function name defined in M, and m is a message containing arguments to fn, with* $fn = m.\text{fname}$.[5]

Below, we will often refer to the m-component of an event as its *context*. Our contract events are *coarse-grained*: they are only identified by entry points and inputs to a specific contract. Event executions of a contract may invoke other contracts, which only return values as parameters. Externally called contracts may modify their own local state, but such external state is not modelled.

Event runs and traces. The *run* of an event e at a contract instance $c = \langle id, M, \rho\rangle$ brings it to a new state ρ', denoted $\rho \xrightarrow{e} \rho'$. A sequence of events is called an *event trace*. An evaluation of contract at instance ρ_0 over an event trace h yielding instance ρ_n is denoted as $\rho_0 \overset{h}{\Longrightarrow} \rho_n = \rho_0 \xrightarrow{e_1} \ldots \xrightarrow{e_n} \rho_n$, which can be obtained by executing all events from h consecutively, updating the blockchain state between steps. Some evaluations may halt abnormally due to a runtime exception, denoted as $\rho \overset{h}{\Longrightarrow} \lightning$. We call such h *invalid traces* (at ρ). Conversely, valid traces evaluate at instance ρ_0 without an exception to a well-formed state ρ_n, which is implicit when we write $\rho \overset{h}{\Longrightarrow} \rho_n$. We now define an event ordering bug.

Definition 2 (Event-ordering bug). *For a contract instance* $c = \langle id, M, \rho_0\rangle$ *and a blockchain state* σ *such that* $\sigma[id] = \rho_0$*, a pair of valid event traces* $h = [e_1, \ldots, e_n]$ *and* $h' = [e'_1, \ldots, e'_n]$ *constitutes an event-ordering bug* iff

- h' *is a permutation of events in h,*
- *if* $\rho_0 \overset{h}{\Longrightarrow} \rho_n$ *and* $\rho_0 \overset{h'}{\Longrightarrow} \rho'_n$*, then* $\rho_n \neq \rho'_n$.

Since we target multi-transactional executions, we use a coarse-grained event model that does not capture reentrancy bugs. Furthermore, for the sake of tractability, the definition is only concerned with a single contract at a time, even though involved events may modify state of other contracts. Therefore, our notion of EO bugs only captures the effects on a *local* state of the contract in focus and won't distinguish between states of other contracts.[6]

[5] In the EVM, function names are not different from other fields of the message, but here we make this separation explicit for the sake of clarity.

[6] One could generalize Definition 2, allowing to catch bugs resulting in transferring money to one beneficiary instead of another, without any local accounting. In this work, we do not address this class of *non-local* EO bugs.

4.2 The Core Algorithm

EthRacer's basic algorithm systematically enumerates all traces of up to a bounded length of k events (configurable), for a given smart contract. One could employ a purely random dynamic fuzzing approach to generating and testing event traces. However, even for a single entry point, different input values may exercise different code paths. In contrast, symbolic execution is a useful technique to reason about each code path, rather than enumerating input values, efficiently. One could consider a purely symbolic approach that checks properties of code paths encoded fully symbolically, to check for output differences. EthRacer uses an approach that combines symbolic analysis of contract code and randomized fuzzing of event trace combinations in a specific way to find EO bugs.

Symbolic event analysis. EthRacer first performs a syntactic analysis to recover the public functions from bytecode of the contract. These functions are externally callable, and thus, these are entry points for each event. Next, it employs standard dynamic symbolic execution technique to reason about the outputs of each event separately. For each event e_i, the analysis marks the event inputs as symbolic and gathers constraints down all paths starting from the entry point of e_i. Modulo implementation caveats (as detailed in Section 5.1), our analysis creates constraints that over-approximate the set of values that drive down a specified path. Note that since the symbolic analysis aims to over-approximate the feasible paths concretely executable under an event, it does not need to always check the feasibility of each path exactly using SMT solvers in this phase. As a result of this analysis, for each event e_i, we obtain a vector of symbolic constraints (as SMT formulae) $\vec{S_i}$ that encode path constraints for the set of execution paths starting from the entry point of e_i. The pairs $\langle e_i, \vec{S_i}\rangle$ is referred to as a *symbolic event*.

Concretization. Conceptually, symbolic events are concretized in two separate steps. The first concretization step is standard in DSE system, where concretization aids path exploration. Specifically, the feasibility of each path constraints in $\vec{S_i}$ using an off-the-shelf SMT solver. We eliminate symbolic path constraints for paths that become infeasible. This concretization makes the symbolic constraints less general, but allows pruning away a lot of false positives due to assuming infeasible paths or values of symbolic inputs.

The second concretization step aims to create one or more concrete value for each path that remains in $\vec{S_i}$. The goal of this step is to enumerate all the traces k concrete events long, and fuzz the contract with these concrete event traces. We use an SMT solver to find value assignments for all symbolic variables in each element of $\vec{S_i}$. These concrete values give us a set of *concrete events* which can be directly executed on the Ethereum Virtual Machine.

Happens-Before relations. EthRacer uses its symbolic analysis to infer a *hb* relationship between symbolic events after the first concretization step. Subsequently, the second concretization step generates concrete events that respect these *hb* relations.

Fuzzing with concrete events. EthRacer does fuzzing with the concrete events output by the previous steps and flags pairs of concretely tested traces that exhibit diverging outputs(*witness pair*) as EO violations. The procedure is explained in section 5.2

4.3 Extracting HB Relations

EVM semantics do not feature programming abstractions like Java's `synchronized` [19] or a rigorous specification of dynamic event precedence [7, 32], which concurrency analysis techniques targeting other languages or platforms use. That said, smart contract developers impose an *intrinsic* ordering on events by engineering ad-hoc synchronization via a mechanism EVM supports natively: exceptions[7] and global variables. Specifically, the contract may set values for global variables in the processing of one event, and later check/use these before performing critical operations under subsequent event. If the check/use of the global variable results in an exception, the second event will not lead to valid trace and the execution of that event will be nullified. For instance, in Example 2 from Section 2, the event $e_1 = \langle \texttt{approve}, ((A, 100), \ldots)\rangle$ always happens before $e_2 = \langle \texttt{transferFrom}, ((O, A, 100), \ldots)\rangle$ since the `require` clause prevents the opposite from happening. Our observation is that if we can detect these cases when there is only one valid ordering between a pair of events, we do not need to test for any event orderings where there is an invalid order.

These ordering relations are captured formally by a happens-before relation [29]. In all event traces, if executing an event e_1 before e_2 leads to an exception, but executing e_2 before e_1 leads to valid (normal) execution, then we say e_2 *happens before* e_1 (denoted $hb(e_2, e_1)$). Finally, if neither from $hb(e_1, e_2)$ and $hb(e_2, e_1)$ holds, e_1 and e_2 can occur in an event trace in any order and are called *independent* events. The precise definition of the notion of happens-before, as used in this paper, is below.

Definition 3 (Happens-before). *For a contract instance c, we say that events e_1 and e_2 are in* happens-before *relation $hb(e_1, e_2)$* wrt. *a set of valid event traces H of c* iff *for any trace $h \in H$, if $h = concat(h_1, [e_2], h_2)$ and $e_1 \in h$, then $e_1 \in h_1$. In other words, e_1 always precedes e_2 in a trace $h \in H$, containing both e_1 and e_2.*

Recovering the complete *hb* relation (as defined above) for a program is difficult even with a powerful symbolic analysis. To address this challenge, our approach infers a "weaker" form of HB-relations, which operates on pairs of events, considering traces with only two events. The following definition makes precise our design choice and enables a direct implementation strategy.

Definition 4 (Weak happens-before). *For a set E of events of a contract instance $c = \langle id, M, \rho\rangle$, two events $e_1, e_2 \in E$ are in a weak happens-before relation ($whb(e_1, e_2)$)* iff *(a) $\rho \xRightarrow{[e_1, e_2]} \rho'$ for some contract state ρ', and (b) $\rho \xRightarrow{[e_2, e_1]} \lightning$.*

The implementation strategy for extracting weak happens-before (WHB) is straight-forward. We execute the two differing orderings of each pair of events in a given trace. If we observe that one order leads to an exception, and other does not, we inductively learn this relationship. We can also identify which functions are independent as per the natural extension of the definition from HB to WHB.

Notice that WHB implies regular HB for traces of two events, but HB for longer traces can include fewer pairs due to state changes made by other events in a trace. Using WHB leads to an under-approximation of the full *hb* relation, as it may render more pairs of events *non-independent*. This could introduce more false negatives (*i.e.*, will miss some EO-bugs), but at the advantage of an exponential reduction in trace combinations to test.

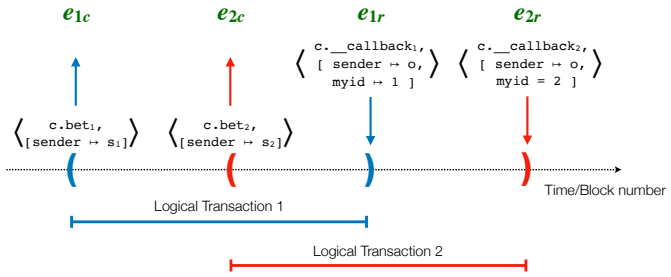


Figure 3: A call/return event trace in `Casino` contract.

An important final optimization helps the symbolic analysis of event pairs. EthRacer builds a weak *hb*-relation for events pairs that have functions which share at least one global variable and at least one of which write to that variable. Only reordering such functions can lead to different values of global variables and thus their corresponding events may be in weak *hb*-relations. EthRacer analyzes the symbolic write sets of program paths and uses this information to guide which events to consider during *hb* analysis.

4.4 Optimizing for Asynchronous Callbacks

This optimization is based on the observation that certain orderings of events are well-paired by EVM semantics. When contract execution adheres to these prescribed paired orderings, the results are what the developer likely intended (and must not be flagged).

At present, EthRacer uses such optimization for asynchronous calls to Oraclize; It is the most popular off-chain service used by Ethereum smart contracts, for off-chain resources such as (*e.g.*, stock exchanges, random number generators). Oraclize handles the query *offline*, by retrieving the data from the required URLs, and later sends it back to the asking contract by calling its function `__callback` in a separate Ethereum transaction.[8] Hence, a call to Oraclize would "return" via an asynchronous callback.

Here, for each `Oraclize` call, there is a matching return or callback. Programmers are expected to match returns to the Oraclize calls, and process accordingly, handling multiple users. The notion that captures intent (or natural program reasoning) is linearizability [22]: each pair of matching call-returns must appear to execute atomically. That is, if two users issue events that trigger Oraclize calls, then either the first user's call-returns are executed before the second user's, or the other way around; if their call-return strictly interleave out-of-order, it leads to an unintended (or unnatural) behavior. This is evident in the `Casino` example (Figure 1).

EthRacer in its fuzzing step recognizes Oraclize calls and asynchronous returns. It first checks all linearizable traces, where matching call-return pairs execute atomically. It memoizes the outputs of these linearizable traces as "canonical" outputs. Then, it generates other possible traces during its testing and reports a witness pair only if the outputs are not equal to all of the canonical outputs. If a trace is found which does not match any linearized event trace, EthRacer flags the contract as having a EO bug, and reports on

[7] In Solidity, exceptions are raised via `throw`, `require` and `assert`.

[8] A contract calling Oraclize must have a function called `__callback`.

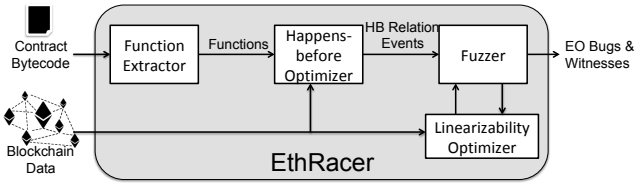


Figure 4: Main components of EthRacer.

the closest linearizable trace in the witness pair. One example of such non-linearizable trace for the `Casino` contract is presented in Figure 3. The concept of linearizable event traces can be extended to other off-chain services which call back into contracts, like Oraclize.

Definition 5 (Linearizable event traces). *A valid event trace $h = [e_1, \ldots, e_n]$ of a contract instance $c = \langle id, M, \rho_0 \rangle$ is linearizable* iff *a state ρ_n, such that $\rho_0 \overset{h}{\Longrightarrow} \rho_n$, can be obtained by executing a trace h', which is a permutation of events in h, such that*

(1) it preserves the order of call/return events from h;
(2) non-overlapping logical transactions in h appear in the same order, as they appear in h.

5 IMPLEMENTATION

EthRacer is implemented in about 6,000 lines of Python and its core components are shown in figure 4. It requires either a contract's bytecode or its Ethereum address as input . If the latter is given, EthRacer assumes the contract is deployed on the main Ethereum blockchain and gets its bytecode and current global storage from the blockchain, and stores it locally. Otherwise, it assumes the contract is not deployed, and thus starts with empty local storage. EthRacer first extracts the functions from contract's bytecode and generates symbolic events. These events are fed to the dynamic symbolic execution engine (Happens-Before Optimizer) for HB analysis. The engine generates concrete events which are then fed to the Fuzzer. The Linearizability Optimizer handles optimizatinon for asynchronous callbacks, described in Section 4.4. EthRacer finally reports all pairs of traces that lead to EO bugs.

5.1 Dynamic Symbolic Execution Engine

All inputs to the entry points of contract bytecode are marked symbolic initially. The symbolic execution starts from the first instruction of the bytecode and interprets sequentially the instructions following the EVM specification [50], with access to running stack, global storage, and local memory. The DSE engine keeps two memory maps: a symbolic map and concrete local map. The local map is initialized with the concrete blockchain state given as input to EthRacer. All global variables of the contract are concretely initialized from the local copy of the global storage. The DSE engine gathers symbolic values, branch, and path constraints in the standard way. We check satisfiability of symbolic constraints using Z3 [12], which can handle operations on numeric and bit-vector domains. Our present implementation supports symbolic analysis of 90% of all EVM opcodes. The unhandled instructions preclude analysis of a small number of paths in 6% of analyzed contracts.

The analysis aims to over-approximate values in its symbolic path constraints as a default strategy. For instance, when checking with path feasibility with an SMT solver, if the solver times out, we assume that the path is feasible. We prune away paths only if the SMT solver returns `UNSAT`. We over-approximate the set of pointer values during the analysis for symbolic memory by not concretizing it eagerly. Similarly, the DSE engine marks return values of `CALL` instructions (which invokes functions from other contracts) as symbolic. During the DSE analysis (and only during this analysis), we do not concretely execute the externally called contract, but the symbolic return value over-approximates the behavior. External calls cannot modify the caller's contract state, hence marking only the return value as symbolic is sufficient.

While our symbolic analysis largely over-approximates, it concretizes in a small number cases which helps improving path coverage. First, when the value of the symbolic variable is needed for checking path feasibility, we lazily concretize it to drive execution down that branch. Second, the DSE engine concretizes lookups for key-value stores (`hashmap`/`hashtable` types), which are called "mappings" in Solidity [47] and implemented using `SHA3` as the hash function. This implementation detail is crucial for finding EO bugs in many contracts, including the ERC-20 example from Figure 2. If the lookup key v is symbolic (*e.g.*, it came as a contract call input), our DSE engine concretizes it to the set of values assigned to it in concrete executions observed during DSE. In addition, the first time a symbolic v is accessed in a write, we assign it a new random concrete value so that the concrete value set is never empty.

Our DSE returns a set of concrete events for functions which affect the shared global state. However, there might be functions for which concrete events may not be generated during the HB analysis. For them, concrete events are generated such that the events execute independently, without throwing an exception.

5.2 Fuzzing Event Traces

Given the set E of all events produced previously, the fuzzer engine creates all possible subsets of E up to a certain size.[9] For each subset, all possible traces, which obey the previously discovered HB relation, are generated. Each such trace is executed on a customized EVM instance, which in comparison to the original EVM (of original Ethereum client) is orders of magnitudes faster, as it runs without performing the proof-of-work mining, and does not participate in a message exchange with other nodes on the network. If EthRacer processes a contract deployed on the main Ethereum blockchain, the customized EVM initially reads its actual global state, and uses this copy for all sequential reads and writes of the global storage.

After executing each trace, the global storage and the balance of the contract are saved. Once all traces for a particular subset have been executed, the tool finds and outputs a list of pairs of EO-bug traces, *i.e.*, pairs of traces that result either in different global storage or in different balances. The tool then performs minimization by creating a set of pairs of *minimal traces* of function calls, reproducing the found EO-bugs. The minimization is done by implementing a simple shrinking strategy: same events are removed from the pair of buggy traces, one by one, while checking whether a

[9] The maximal size is parametrized, thus it can be increased or reduced; we tried sizes from 2 to 6 in our experiments.

```
contract ERC721  {
  function addPermission(address _addr) public
    onlyOwner { allowPermission[_addr] = true; }
  function removePermission(address _addr) public
    onlyOwner { allowPermission[_addr] = false; }...}
```

Figure 5: False positive: fragment of an ERC721 contract.

"smaller" pair of traces still constitute an EO-bug. In our experience, the size of the set of minimal traces is significantly smaller than the length of the full list of buggy traces, so having the minimization procedure provides better user experience and allows for faster confirmation of witnesses by a human.

6 EVALUATION

We evaluate EthRacer to measure how many contracts are flagged as having EO bugs, *i.e.*, how many show differing outputs under different orderings of events. Further, we measure how much effort an analyst has to spend per contract to analyze the flagged traces. We also assess how EO bugs compare to transaction re-ordering bugs checked by the latest version of Oyente tool [40]. We highlight only important findings in this section and provide more case studies in appendices given as supplementary material. We urge the readers to refer to them to gain better insights into our results.

Evaluation subjects. Around 1% of the smart contracts deployed on Ethereum have source code. EthRacer can directly operate on EVM bytecode and does not require source. To test the effectiveness of the tool, we select 5,000 contracts for which Solidity source code[10] is available and another 5,000 contracts randomly chosen from the Ethereum blockchain (for which source is unavailable).

Apart from these contracts, for analysis of contracts supporting asynchronous callbacks, we find 1,152 unique contracts on the Ethereum blockchain to which Oraclize callbacks have occurred. We filter out contracts with less than two Oraclize queries and are left with 423 unique contracts out of which 154 have Solidity source code available. We analyze these 423 contracts,

Experimental Setup. We run all our experiments on a Linux server, with 64GB RAM and 40 CPUs Intel(R) Xeon(R) E5-2680 v2@2.80GHz. We process the contracts in parallel on 30 cores, with one dedicated core per contract. We set a timeout of 150 minutes per contract. EthRacer is configured to output 3 pairs of events for each pair of functions in the HB relation from its an analysis component; the timeout per pair of functions in the *hb* relation is 2 minutes. For each function with no event generated in the *hb* creation phase, EthRacer either generates one event or times out in 1 minute. These events are used for the fuzzing step in EthRacer. The outputs of the EthRacer are a set of pairs of event traces, flagged as having EO bugs. EthRacer is publicly available on GitHub.[11]

6.1 Efficacy of EthRacer

EthRacer flags a total of 836 (8%) EO violations which includes 47 unmatched callback violations in the analyzed contracts, holding hundreds of millions of dollars worth of Ether over their lifetime.

Out of the 10,423 analyzed contracts, 836 are flagged as having EO bugs. 47 contracts out of 423 analyzed are buggy due to asynchronous callbacks. Currently, 28 of them are live and are transaction-intensive overall, as they have handled 55,946 transactions in their lifetime. Among the remaining 789 contracts which are flagged for on-chain transaction ordering, 674 have source code, which we manually analyze, and 115 are from the set of contracts without source. At present, 785 out of 789 contracts are live and the volume of processed transactions over their lifetime amounts to 1,649,192 transactions on the public Ethereum blockchain.

6.2 Manual Analysis Effort

EthRacer outputs a set of pairs of traces exhibiting EO bugs. As shown in Figure 6c, for EO bug-flagged contracts, EthRacer produces only a few minimized witness pairs, on average two, with the majority having only a single pair. Similarly, the contracts with callback-related bugs have only one witness pair for all vulnerable contracts. Accordingly, manual post-hoc analysis effort is minimal.

To confirm the simplicity of the post-hoc analysis and to determine whether the EO violations are intentional (benign) or buggy, we manually analyze the flagged cases. We take a 127 randomly chosen contracts flagged for EO violations, of which 27 contracts (with source code) were flagged for unmatched callbacks. For each of the inspected contracts, we need only a few minutes to check whether the EO violation is true bug or likely benign.

On-chain transaction ordering. For the 100 contracts we analyzed, 52% are closer to the examples presented in Section 2 and we adjudged them to be true bugs. Apart from the example presented in Section 2, we find several other examples of subtle true EO violations. Two additional examples of such contracts named `Contest` and `Escrow`, are presented in supplementary material.

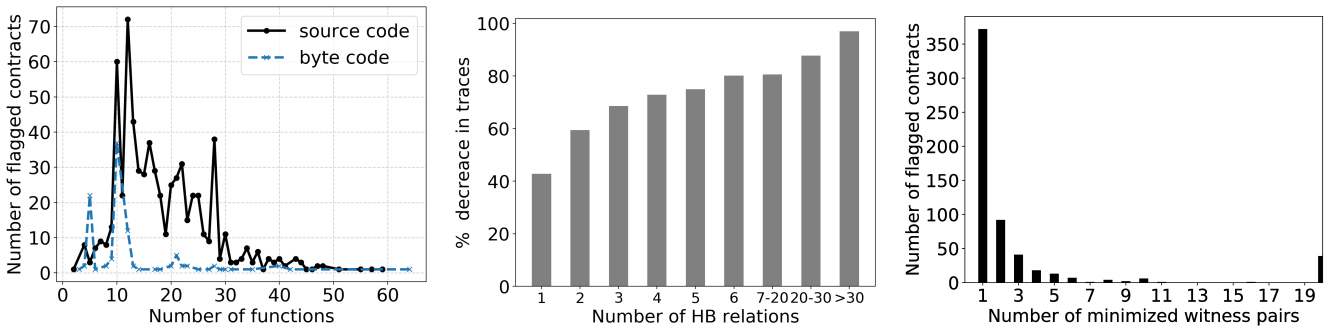
We deemed that 48% of the 100 analyzed contracts were benign violations of the EO definition. These contracts were straightforward to analyze, as they have a small set of global variables shared between functions of the traces which take different values upon reordering. Consider the ERC721 contract shown in Figure 5, where `addPermission` and `removePermission` are called by the contract's owner to update the common variable `allowPermission`.

Off-chain asynchronous callbacks. For these violations, among the 47 flagged contracts, 27 have Solidity source code, which we manually analyze. 23 of these contracts have true and subtle EO violations, which follow two predominant patterns:

- **Unprocessed `_queryId`:** The first class occurs in contracts that do not process `_queryId` in their `__callback` function. The contracts assume Oraclize is synchronous, *i.e.*, assume that each query to Oraclize is immediately followed by a reply from Oraclize. The contract `BlockKing` first analyzed by Sergey and Hobor [43], is an example of such a contract. We have identified one more contract, called `Gamble`, that suffers from a similar mistake.
- **Improper check on Ether:** This second class of bugs is more subtle and has not been identified in prior works. It occurs in contracts similar to our `Casino` example from Section 2.

Out of 27 contracts flagged for unmatched callbacks, 4 contracts have EO violations which seem to be benign (or intended logic) in contracts. Specifically, they have two or more traces that differ in output dependent on the timestamp of the mined block.

[10] We obtain contract source codes from the popular website Etherscan [2].
[11] https://github.com/ashgeek/Ethracer



(a) # functions in flagged contracts. **(b) Impact of the # HB-related events pairs.** **(c) # minimal traces in flagged contracts.**

Figure 6: (a) shows that a majority of flagged contracts have $10-30$ functions, (b) demonstrates the reduction in traces to fuzz, with increasing number of HB-related event pairs and (c) shows that EthRacer outputs only $1-3$ witnesses for most contracts.

6.3 Performance

About 95% of the contracts require 18.5 minutes of analysis time with EthRacer on average per contract, whereas only 5% of them timeout after 150 minutes. Of the 18.5 minutes, EthRacer spends about 15 minutes to produce events and 3.5 minutes for fuzzing.

Figure 6a provides a quantitative explanation for this efficiency. It shows the total number of functions in contracts; one can see that a large number of contracts have over 10 callable functions, thus analyzing all permutations would lead to prohibitively large number of traces to test concretely.

Figure 6b shows the correlation between the size of the *hb* relation set and the number of potential traces created by the fuzzer. As shown in the figure, an increase in the number of *hb* relations leads to a decrease in the number of possible traces the fuzzer needs to create and check. For instance, on average of all analyzed contracts, having a single *hb* relation reduces the number of possible traces by nearly 42%. Moreover, the average number of HB relations produced by EthRacer for a contract is nearly 5.5.

An illustrative example. EthRacer finds the ERC-20 bug mentioned in Section 2 in 5 minutes, producing a single minimized pair of traces to inspect. The tool first collects all 11 functions of the contract and filters 8 of them leaving only for 3 functions for re-ordering. Without this optimization, our *hb* analysis (Section 4.3) would inspect $\binom{11}{2}$ or 55 pairs instead of 3. After 4 minutes of symbolic analysis to recover *hb* relations, EthRacer creates 7 concrete events as input to the fuzzer. In 1 minute, the fuzzer analyzes traces of length ranging from 2 to 6 and it analyzes a total of 2,560 traces. This number would be 8,652 without our *hb* relation analysis, which has over $3x$ improvement to concrete fuzzing in this example. After fuzzing, EthRacer outputs 43 traces which have bugs and the minimization step produces a single pair of traces to inspect. This is precisely the pair presented in Section 2.

6.4 Comparing EthRacer and Oyente

Oyente is a symbolic execution analyzer which flags contracts with two different traces having different Ether flows (*i.e.*, changes in a contract's balance) [30, 40]. It tries to produce contexts m_1 and m_2 such that traces (e_1, e_2) and (e_2, e_1) result in two different ether flows where $e_1 = \langle f_1, m_1 \rangle$ and $e_2 = \langle f_2, m_2 \rangle$. Here, f_1 and f_2 can be the same function. It uses our aforementioned baseline strategy of enumerating all pairs (traces of depth two) and symbolically checking if the two paths lead to two different balances.

Our definitions of event ordering (EO) may appear similar to transaction ordering dependency bugs (TOD) defined and analyzed by the Oyente tool. There is, however, several definitional and technical differences between the two tools:

(1) Oyente does not reason about unmatched callbacks; TOD bugs are a strict subset of on-chain EO violations.
(2) Oyente detects two paths with different **send** instructions, which, unlike EO bugs, are characterized by differences in the final output states. Oyente does not account for global state changes in a transaction other than account balances. Thus, TOD detections by Oyente may include two traces sending to the same recipient, causing no modifications to state other than balances.
(3) Oyente checks for differences between pairs of traces to prevent combinatorial explosion in path space analysis. EthRacer can analyze any combination of traces; we chose a limit of 6 in our experiments. The key reason for its scalability is the use of its HB-relation analysis and analysis optimizations.
(4) Oyente only reports symbolic path constraints responsible for TOD, without giving concrete inputs which exhibit the bug; this is unlike EthRacer which produces concrete and minimized event traces for the analyst to inspect.

Experimental Comparison. To enable a manual analysis of results, we compared EthRacer and Oyente on 5,000 contracts with available Solidity source code with Oyente and EthRacer. First, Oyente has no notion of linearizability and thus it detects none of contracts with unmatched callbacks that EthRacer flags. For on-chain transaction ordering, EthRacer flags 674 contracts, while Oyente flags 251 cases when its internal default timeout is set to 150 minutes (same as EthRacer) and in online mode.[12] Also, Oyente terminates for all tested contracts in this timeout.

Out of the 251 contracts, 78 are flagged by EthRacer as well. We manually inspect all the remaining 173 contracts flagged by Oyente. We find that all these cases are false EO violations–this

[12] In the online mode, Oyente reads the values of the global variables from the actual blockchain, similar to EthRacer in its default mode.

confirms that EthRacer finds all of true TOD violations flagged by Oyente, and finds more than double of EO bugs which extend beyond TOD bugs as well. Few additional case studies which only EthRacer finds, apart from the ones in Section 2 are given in the supplementary material. Our manual analysis finds that Oyente flags TOD false positives mainly for two reasons. First, Oyente assumes that different Ether flows always send Ether to different addresses, which is often not the case. Second, in contracts with zero balance, all Ether flows can only send zero Ether, thus they do not differ in output balances although Oyente flags them incorrectly.

7 RELATED WORK

Our work generalizes the class of bugs arising due to non-deterministic scheduling of concurrent transactions. The work by Luu *et al.* [30] identified a specific kind of on-chain EO violations related to balance updates and checked for differences across only a pair of paths. Our work extends this to longer traces and full state differences, finding nearly 4*x* more true EO violations of previously unknown bugs. Our class of bugs are unrelated to liveness or safety properties identified more recently by symbolic execution techniques [26, 27, 38]. At a technical level, these works provide robust symbolic execution systems which is equivalent to our assumed starting baseline technique. However, in this work, our techniques are complementary to the baseline, directly addressing the combinatorial blowup due to checking traces of large lengths. To show this empirically, we compared EthRacer to the open-source Oyente that operates on bytecode, in Section 6.4. Alternative systems such as Zeus [26] could be used instead of Oyente, but Zeus is not publicly available and operates on Solidity source rather than bytecode.

Sergey and Hobor assert that many issues stemming from the nondeterministic transactional nature of smart contracts are analogous to well-studied bug classes in shared-memory concurrency [43]. For instance, TOD is a kind of *event race* [15, 41], while DAO and mishandled responses from asynchronous callbacks service are a violation of logical atomicity [21, 22]. Knowledge of some of those issues has grown in the community [11], but we are not aware of any other principled tool that detects them at scale.

The most related work to our approach to finding unmatched callbacks is the dynamic analysis by Grossman *et al.* [21] for detecting of DAO-like re-entrancy vulnerabilities [13, 45]. Their work checks for a specific linearizability property of *Effectively Callback Freeness* (ECF): a contract c is ECF *iff* for any transaction involving a reentrant call to c one can construct an equivalent transaction without reentrant calls. Their dynamic analysis employs the notion of *Invocation Order Constraint* (IOC) graph, which only captures the *fine-grained* shape of a contract execution within a *single* transaction. Verifying ECF dynamically is drastically simpler than challenge tackled by EthRacer, as ECF entails checking the commutativity reads/writes of functions under a *single* transaction execution. In this paper, we are interested in handling multi-transactional executions and the associated combinatorial blowup associated. Our notion of *hb*-relations and our dynamic symbolic analysis are entirely different. Consequently, EthRacer finds 47 callback-related errors and 674 on-chain EO violations in about ten thousand contracts, none of which are reported in the work by Grossman *et al.* [21]. Their work finds 9 contracts which are vulnerable to re-entrancy bugs on the Ethereum public blockchain, which fall outside the goals and definition of EO violations.

Concurrent work on the Servois [5] tool for automatically generating commutativity conditions from data-structure specifications has been successfully used to confirm the presence of a known linearizability violation in a simple Oraclize-using contract `BlockKing` [43]. However, Servois can only work with an object encoded in a tailored domain-specific language, thus, it cannot be applied for automatically detecting bugs at scale of an entire blockchain.

Other approaches based on symbolic analysis of Solidity code or EVM implementations have been employed to detect known bugs from a standard "smart contract vulnerability checklist" [4, 14, 31] in tools, such as Mythrill [34, 35], SmartCheck [46], Securify [49], and Manticore [33], none of which addressed EO bugs.

Our work shares similarity to a vast body of research on checking linearizability and data race detection in traditional programming languages [7, 8, 10, 17, 32, 36]. One might expect that some of those tools could be immediately applicable for the same purpose to smart contracts. The main obstacle to do so is that, unlike existing well-studied concurrency models (sequential consistency [28], Android [7], *etc*), Ethereum contracts do not come with the formally specified model of explicit synchronization primitives or have explicit programming abstractions. Because of this, our procedure for inferring intrinsic *hb* relations is considerably different from prior works. We believe our approach lays out useful abstractions for future works investigating concurrency in smart contracts.

While our analysis is designed to detect concurrency-related bugs at scale, a complementary approach would be to mechanically verify a contract's implementation to adhere to the desired atomicity/race freedom properties. At the moment, a number of efforts resulted in a complete mechanisation of EVM semantics [50] in various frameworks for interactive and automated proofs: F* [6, 20], Isabelle/HOL [3, 24], Coq [25], and K [23, 42]. It should be possible to encode our properties of interest on top of those semantics, allowing for the proofs similar to what has been done before for concurrent objects [16, 44], providing the ultimate safety guarantees. In this regard, we consider our tool to be complementary to those future efforts, filling the niche modern race detectors occupy for efficiently finding bugs in deployed concurrent applications [7, 9, 18].

8 CONCLUSION

We have studied event-ordering bugs in Ethereum smart contracts by exploiting their similarity to two classic notions in concurrent programs: linearizability and synchronization violations. We have provided a formal model for these violations. We have shown how to infer intrinsic happens-before relations from code and demonstrated how to use such relations to shrink the bug search space.

ACKNOWLEDGEMENTS

We thank Shweta Shinde and Shiqi Shen for their valuable comments and their help with writing a previous version of this paper. We thank sponsors of the Crystal Center at National University Of Singapore which has supported this work. Further, Ivica Nikolic is supported by the Ministry of Education, Singapore under Grant No. R-252-000-560-112. Aquinas Hobor was partially supported by Yale-NUS College grant R-607-265-322-121.

REFERENCES

[1] 2018. Ethereum Github. https://github.com/ethereum/EIPs/issues/738. Accessed: 2018-05-05.

[2] 2018. Etherscan. https://etherscan.io. Accessed: 2018-05-05.

[3] Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.

[4] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST (LNCS)*, Vol. 10204. Springer, 164–186.

[5] Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *TACAS (Part I) (LNCS)*, Vol. 10805. Springer, 115–132.

[6] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.

[7] Pavol Bielik, Veselin Raychev, and Martin T. Vechev. 2015. Scalable race detection for Android applications. In *OOPSLA*. ACM, 332–348.

[8] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 2018. RacerD: Compositional Static Race Detection. *PACMPL* OOPSLA (2018).

[9] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *POPL*. ACM, 651–662.

[10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. 2010. Line-up: a complete and automatic linearizability checker. In *PLDI*. 330–340.

[11] ConsenSys Inc. 2018. Ethereum Smart Contract Security Best Practices: Known Attacks. https://consensys.github.io/smart-contract-best-practices/known_attacks/

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

[13] Michael del Castillo. 2016. The Dao attack. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/ 16 June 2016.

[14] Kevin Delmolino, Mitchell Arnett, Ahmed E. Kosba, Andrew Miller, and Elaine Shi. 2016. Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab. In *FC 2016 International Workshops (LNCS)*, Vol. 9604. Springer, 79–94.

[15] Dimitar Dimitrov, Veselin Raychev, Martin T. Vechev, and Eric Koskinen. 2014. Commutativity race detection. In *PLDI*. ACM, 305–315.

[16] Thomas Dinsdale-Young, Pedro da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 2017. Caper - Automatic Verification for Fine-Grained Concurrency. In *ESOP (LNCS)*, Vol. 10201. Springer, 420–447.

[17] Cormac Flanagan and Stephen N. Freund. 2009. FastTrack: efficient and precise dynamic race detection. In *PLDI*. ACM, 121–133.

[18] Cormac Flanagan and Stephen N. Freund. 2010. The RoadRunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM, 1–8.

[19] Brian Goetz, Tim Peierls, Joshua J. Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice*. Addison-Wesley.

[20] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269.

[21] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018), 48:1–48:28.

[22] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. 12, 3 (1990), 463–492.

[23] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Daejun Park, Yi Zhang, Brandon Moore, and Grigore Rosu. 2018. KEVM: A Complete Semantics of the Ethereum Virtual Machine. In *CSF*. IEEE. To appear.

[24] Yoichi Hirai. 2017. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *1st Workshop on Trusted Smart Contracts (LNCS)*, Vol. 10323. Springer, 520–535.

[25] Yoichi Hirai. 2017. Ethereum Virtual Machine for Coq (v0.0.2). Published online on 5 March 2017. https://medium.com/@pirapira/ethereum-virtual-machine-for-coq-v0-0-2-d2568e068b18

[26] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing Safety of Smart Contracts. In *NDSS*.

[27] Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security*.

[28] Leslie Lamport. 1978. The Implementation of Reliable Distributed Multiprocess Systems. *Computer Networks* 2 (1978), 95–114.

[29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[30] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.

[31] Richard Ma, Steven Stewart, Vajih Montaghami, Ed Zulkoski, and Leonardo Passos. 2017. Quantstamp : The protocol for securing smart contracts. https://quantstamp.com/

[32] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. 2014. Race detection for Android applications. In *PLDI*. ACM, 316–325.

[33] Manticore 2018. Manticore: A symbolic execution tool for analysis of binaries and smart contracts. https://github.com/trailofbits/manticore

[34] Bernhard Mueller. 2018. How Formal Verification Can Ensure Flawless Smart Contracts. https://media.consensys.net/how-formal-verification-can-ensure-flawless-smart-contracts-cbda8ad99bd1

[35] Mythril 2018. Mythril: A security analysis tool for Ethereum smart contracts. https://github.com/b-mueller/mythril

[36] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *PLDI*. ACM, 308–319.

[37] Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf

[38] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. *CoRR* abs/1802.06038 (2018).

[39] Oraclize 2016. Oraclize – Blockchain Oracle service, enabling data-rich smart contracts. http://www.oraclize.it.

[40] Oyente 2018. Oyente: An Analysis Tool for Smart Contracts. https://github.com/melonproject/oyente

[41] Veselin Raychev, Martin T. Vechev, and Manu Sridharan. 2013. Effective race detection for event-driven programs. In *OOPSLA*. ACM, 151–166.

[42] Grigore Rosu. December 2017. ERC20-K: Formal Executable Specification of ERC20. https://runtimeverification.com/blog/?p=496

[43] Ilya Sergey and Aquinas Hobor. 2017. A Concurrent Perspective on Smart Contracts. In *1st Workshop on Trusted Smart Contracts*.

[44] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized Verification of Fine-grained Concurrent Programs. In *PLDI*. ACM, 77–87.

[45] Emin Gün Sirer. 2016. Reentrancy Woes in Smart Contracts. http://hackingdistributed.com/2016/07/13/reentrancy-woes/ 13 July 2016.

[46] SmartCheck 2018. SmartCheck. https://tool.smartdec.net/

[47] Solidity 2016. Solidity: A contract-oriented, high-level language for implementing smart contracts. http://solidity.readthedocs.io

[48] Nick Szabo. 1996. Smart Contracts: Building Blocks for Digital Markets.

[49] Petar Tsankov, Andrei Marian Dan, Dana Drachsler Cohen, Arthur Gervais, Florian Buenzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. *CoRR* abs/1806.01143 (2018).

[50] Gavin Wood. 2014. Ethereum: A Secure Decentralised Generalised Transaction Ledger. https://ethereum.github.io/yellowpaper/paper.pdf