Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing

Mingxi Ye
Sun Yat-sen University
Guangzhou, China
yemx6@mail2.sysu.edu.cn

Yuhong Nan
Sun Yat-sen University
Guangzhou, China
nanyh@mail.sysu.edu.cn

Zibin Zheng*
Sun Yat-sen University
Guangzhou, China
zhzibin@mail.sysu.edu.cn

Dongpeng Wu
Sun Yat-sen University
Guangzhou, China
wudp8@mail2.sysu.edu.cn

Huizhong Li
Webank
Shenzhen, China
wheatli@webank.com

ABSTRACT

Decentralized applications (DApps) consist of multiple smart contracts running on Blockchain. With the increasing popularity of the DApp ecosystem, vulnerabilities in DApps could bring significant impacts such as financial losses. Identifying vulnerabilities in DApps is by no means trivial, as modern DApps consist of complex interactions across multiple contracts. Previous research suffers from either high false positives or false negatives, due to the lack of precise contextual information which is mandatory for confirming smart contract vulnerabilities when analyzing smart contracts.

In this paper, we present *IcyChecker*, a new fuzzing-based framework to effectively identify State inconsistency (SI) Bugs – a specific type of bugs that can cause vulnerabilities such as re-entrancy, front-running with complex patterns. Different from prior works, *IcyChecker* utilizes a set of accurate contextual information for contract fuzzing by replaying the on-chain historical transactions. Besides, instead of designing specific testing oracles which are required by other fuzzing approaches, *IcyChecker* implements novel mechanisms to mutate a set of fuzzing transaction sequences, and further identify SI bugs by observing their state differences. Evaluation of *IcyChecker* over the top 100 popular DApps showed it effectively identifies a total number of 277 SI bugs, with a precision of 87%. By comparing *IcyChecker* with other state-of-the-art tools (i.e., Smartian, Confuzzius, and Sailfish), we show *IcyChecker* not only identifies more SI bugs but also with much lower false positives, thanks to its integration of accurate on-chain data and unique fuzzing strategies. Our research sheds light on new ways of detecting smart contract vulnerabilities in DApps.

*Zibin Zheng is the corresponding author.

ISSTA '23, July 17–21, 2023, Seattle, WA, United States

ACM ISBN 979-8-4007-0221-1/23/07…$15.00
https://doi.org/10.1145/3597926.3598057

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Decentralized Application; Smart Contract; Vulnerability detection; Fuzz Testing

ACM Reference Format:
Mingxi Ye, Yuhong Nan, Zibin Zheng*, Dongpeng Wu, and Huizhong Li. 2023. **Detecting State Inconsistency Bugs in DApps via On-Chain Transaction Replay and Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, United States.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598057**

1 INTRODUCTION

Decentralized applications (DApps) consist of multiple smart contracts running on Blockchain [3]. With a set of unique features such as decentralization and full anonymization, DApps have been widely adopted in various scenarios including financial services [47], Arts and collectibles [46], and gaming [32]. As of Oct. 2022, there are more than 1,500 DApps in Ethereum, with around 30,000 users. The total value of transactions in this ecosystem reaches nearly five billion daily [12]. Given the increasing popularity, the security of such DApps is also of significant importance. For example, early in 2016, the well-known DAO attack [31] leads to about $50M loss. In the next following years, other popular DApps (e.g., DAO Maker, Parity, bZx) are also impacted with considerable financial losses by different severe vulnerabilities, such as front-running [24], access control [49], and bad randomness [15].

State inconsistency bugs. State inconsistency (SI) bugs are a class of vulnerabilities that allow an adversary to manipulate global states of smart contracts (storage variables, block states, etc.), hence making such states inconsistent with the expectations of the victim (e.g., the DApp user). Exploiting SI bugs, the adversary can arbitrarily invoke any legitimate but malicious functions in smart contracts (as transactions), and further influence a targeted transaction to gain financial benefits.

In fact, a large portion of smart contract vulnerabilities is caused by SI bugs. For example, in front-running attacks [24], the attacker inserts an additional transaction (external function call) in front of the targeted transaction in the same batch (i.e., a transaction

block). As another example, in re-entrancy attacks [50], the attacker hijacks the original control flow of a legitimate transaction and invokes function calls within the target transaction. In both of these two attacks, the attacker successfully influences critical states of smart contracts and further causes biased results to the DApp or victim app users. More recently, reports showed that the attacker also exploits SI bugs to violate access control logic and perform more complex front-running attacks, in which the access of specific functions relies on particular transient state variables [49].

Challenges in detecting SI bugs. Compared with existing works that only consider vulnerabilities within a single smart contract (e.g., Oyente [28]), identifying SI vulnerabilities heavily relies on cross-contract analysis that is well-known to be non-trivial [29]. More specifically, a DApp may expose several public interfaces as its entry point (i.e., ABI) for cross-contract interactions. Therefore, those security-sensitive states of the DApps can be affected by an arbitrary invocation from other contracts under different contexts (i.e., invocation timing, parameter values, etc), hence, resulting in SI bugs. While a series of frameworks have been proposed to detect cross-contract vulnerabilities [25, 50, 51], these works share several inherent limitations in detecting SI bugs. Particularly, we summarise two key challenges in prior works: **(1) High false alarms due to the lack of fine-grained contextual information.** As mentioned earlier, static analysis is fundamentally limited to accurately recovering the full contextual information of contract interactions at runtime. Therefore, these approaches tend to over-approximate the contextual information, hence resulting in high-false positives. More specifically, with the exposed public interfaces, the state(s) of a specific smart contract can be affected by any other smart contract and any function invocation. A vulnerability may exist only when the smart contract is invoked by a particular account (e.g., the owner account). To this end, static analysis approaches have to blindly consider all possible scenarios and report all the results, which might not be feasible in practice. As shown by prior research [50], tools such as Oyente [28] and Securify [45] have an average number of 85% false alarms in detecting re-entrancy vulnerability, a subset of SI bugs. **(2) High false negatives due to the lack of effective testing oracles.** As an alternative solution, fuzzing-based approaches, such as Smartian [7] and ILF [18], dynamically emulate smart contracts with more accurate contextual information for vulnerability detection. However, these approaches heavily rely on already-known attack patterns to construct effective testing oracles [21]. In practice, as we will further show in Section 2, for many attack patterns in SI bugs, it is extremely difficult to come up with sound testing oracles, not to mention effectively identifying other SI bugs with new attack patterns.

Our work. In this paper, we propose *IcyChecker*, a new framework to detect SI bugs by replaying and fuzzing historical transactions with precise fine-grained contextual information. To address the above limitations in prior works, *IcyChecker* is featured with three phases, namely, (1) Contextual-information collection (CIC) by replaying the historical on-chain transactions, (2) Transaction sequence generation (TSG) for contract fuzzing, and (3) Transaction sequence mutation (TSM) for detecting SI bugs.

Firstly, different from prior research which emulates transactions with fully offline analysis, *IcyChecker* faithfully replays the on-chain transactions and records the corresponding state changes in each transaction. In this way, it precisely extracts the real-world transaction states and other relevant contextual information (e.g., related variables and the potential relevant external contracts). With such more fine-grained contextual information, *IcyChecker* can perform more targeted fuzzing to trigger real SI bugs with high probability.

Then, with the collected information (transactions and historical states) of real-world on-chain data, *IcyChecker* selectively generates a set of feasible transaction sequences (cross-contract invocations) with accurate contextual information for fuzzing. To detail, *IcyChecker* finds other related addresses (such as callers and callees of the historical transactions) and generates new valid transaction sequences for fuzz testing. More importantly, when replaying these transactions, *IcyChecker* takes their exact on-chain states as the input. With this accurate contextual information, it is closer to the real-world execution environment for finding SI bugs.

Further, instead of designing ad-hoc testing oracles to identify SI bugs [21] that can easily bring false alarms or false negatives, *IcyChecker* identifies SI bugs based on observing state changes of different mutated transaction sequences. More specifically, *IcyChecker* generates a set of *state-consistent transaction sequences* (see Section 4.2 for more details) through specific mutation strategies (i.e., transaction reordering, transaction insertion, and block information update). By *state-consistent transaction sequences*, we meant the final states of the mutated transaction sequence *should* be the same as its original sequence. With this criteria, *IcyChecker* checks whether any security-critical states of a DApp are indeed affected by such transaction mutation in an unexpected manner. If a mutated transaction sequence ends with a different final state(s) (i.e., causing state inconsistency), we consider the original transaction sequence to be affected by an SI bug.

To evaluate the effectiveness of *IcyChecker*, we analyzed the top 100 DApps from DappRadar [11], one of the most popular app stores for DApps. We collected the latest thousand transactions of these DApps and further constructed and mutate various sequences to detect SI bugs in these DApps. In total, *IcyChecker* successfully identified 277 SI bugs. Our manual inspection of these reported bugs showed *IcyChecker* achieves a precision of 87%. Particularly, 45 of the reported bugs can lead to vulnerabilities, and the rest are related to fairness issues. Among the confirmed SI bugs, nine of them have new attack patterns which are never disclosed by prior research. We have reported these bugs to corresponding authorities[1], and three of them have already been confirmed. Besides, we compare *IcyChecker* with Smartian [7], Confuzzius [44], and Sailfish [1], three state-of-the-art frameworks which can be used for detecting SI bugs. By analyzing the same set of DApps, we show that *IcyChecker* successfully identifies 239, 228, and 238 more SI bugs than Smartian, Confuzzius, and Sailfish, respectively. In addition, due to the accurate contextual information considered by *IcyChecker* in its fuzzing process, the false positives of *IcyChecker*

[1]We report the identified bugs to CNVD [8], the National Vulnerability Database of China.

are also significantly lower than these frameworks (see Section 6.4 for more details).

In the spirit of open science, we will release the artifact of *IcyChecker*, as well as the corresponding dataset used in our research. A preliminary version of *IcyChecker* and part of our dataset (i.e., the DApp and their corresponding contract addresses, as well as the historical transactions used for fuzzing) can be accessed via the following anonymized repository: https://github.com/MingxiYe/IcyChecker-Artifact.

The contributions of this paper can be summarized as follows:

- We propose a novel framework to detect State Inconsistency (SI) bugs based on transaction-based fuzzing and differential analysis. To the best of our knowledge, *IcyChecker* is the first of its kind to utilize the on-chain contextual information for smart contract fuzzing.
- We propose a set of transaction mutation strategies, which enable *IcyChecker* to effectively discover SI bugs without relying on testing oracles as in prior approach [7, 21, 33, 39].
- We conduct extensive experiments to evaluate the effectiveness of *IcyChecker*. Results showed that *IcyChecker* outperforms other state-of-the-art tools by identifying more SI bugs with much lower false positives.
- We make *IcyChecker* and its dataset public available to benefit future research in the community.

The rest of this paper is organized as follows: Section 2 gives some background knowledge and a motivating example to show the key idea of *IcyChecker*. Section 3 summarizes the key patterns of SI bugs and gives an overview of *IcyChecker*. Section 4 elaborates on the detailed design of *IcyChecker* and Section 5 shows implementation details. Section 6 presents the experimental setup and evaluation of *IcyChecker*. Section 7 discusses the key advantages of *IcyChecker* via a case study, as well as its limitations. Section 8 discusses related work and Section 9 concludes the paper.

2 BACKGROUND AND MOTIVATION

2.1 Background

Smart contract and transaction. In the Ethereum blockchain, each smart contract is an executable program running on top of the Ethereum Virtual Machine (EVM). Currently, most DApps consist of multiple smart contracts. Every smart contract has one or more public functions as its entry point, allowing itself to be invoked by other smart contracts. In addition, smart contracts use addresses as identities to invoke each other [48]. Participants of DApps, such as users and other DApps, typically use transactions to perform cross-contract communication, such as transferring Ethereum tokens (i.e., Ether), executing a specific smart contract function, or deploying a new smart contract.

Contract state. In smart contracts, states are referred to as the storage area of a specific smart contract or transaction. Each smart contract maintains a set of persistent data as storage variables. For example, users' balance of token contracts, token price of financial markets, and owner's address of privileged contracts. As shown in Figure 1, by executing transactions, it actually modifies state variables of related smart contracts. More importantly, such state changes are permanently recorded and updated on-demand by the

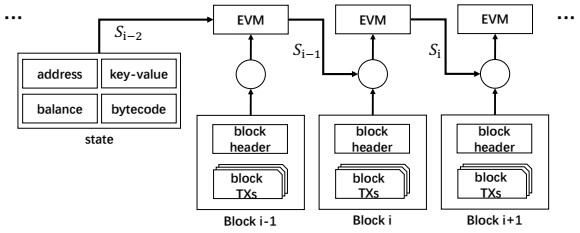


Figure 1: The state transition procedure of blockchain.

Ethereum blockchain. For example, the transfer of ether tokens from one to another will result in the change of `balance` – a critical state in most smart contracts. As we will further show, many states in smart contracts are security-critical. If such states are changed by an adversary unexpectedly (e.g., hijacking the control flow of a running transaction [17]), we consider the smart contract vulnerable to state inconsistency bug.

2.2 Motivating Example

As a motivating example, Figure 2 shows a simplified code snippet of two smart contracts in a real-world DApp (i.e., `Inverse Finance`). The DApp allows its users to pledge one kind of asset, borrow another, and exchange distinct assets. In this case, users pledge `cWBTC` token to borrow `WBTC` token. This DApp is vulnerable to an SI bug in which the attacker can borrow more desired tokens than the value of her pledges by sending malicious transactions. More specifically, when a user wants to borrow a specific type of token, the `borrowAll()` function checks the borrower's pledges by querying the transient price of `cWBTC` token (line 7 in Figure 2(a)). Unfortunately, the price of `cWBTC` token depends on a critical global state (the balance of `WBTC`) in another contract (i.e., Exchange), which can be easily manipulated by an adversary from outside. To perform the attack, the attacker can quickly swap a significant amount of another token to `WBTC`, and hence, increase the market balance of `WBTC` before invoking function `borrowAll()`. As a result, the `cWBTC` price calculated by the function `latestAnswer()` (line 12 in Figure 2(a)) increases as well, and the attacker can borrow more assets than normal.

2.3 Limitations of Existing Works

Existing methods (i.e., both static and dynamic analysis) can not effectively identify such SI bugs due to the following reasons.

Missing fine-grained contextual information. The root cause of the SI bug in `cWBTC` is caused by using an unsafe external state variable (i.e., `WBTC.balanceOf(Exchange)`) (line 13 in Figure 2(a)) for token price calculation. However, static analysis such as Sailfish [1] can not consider the actual values of contract states (e.g., the callers of related transactions, the amount of swapped tokens, and actual price changes). Without such fine-grained contextual information, it is impossible to identify the SI bug in this example.

Lack of effective testing oracles. Runtime fuzzing-based approaches (e.g., xFuzz [51]) can dynamically emulate the execution of the two smart contracts and obtain the runtime values. However, in most cases, the price changes of a specific token are quite normal in blockchain. As a result, there is no effective testing oracle to

```
1  contract cWBTC {
2      /* record of cWBTC balances for each account */
3      mapping (address => uint) internal accountTokens;
4      uint price;     /* price of cWBTC */
5
6      function borrowAll(address borrower) external
           returns (bool) {
7          sumCollateral = accountTokens[borrower] *
               latestAnswer();
8          doTransferOut(borrower, sumCollateral);
9          ...
10     }
11
12     function latestAnswer() public view return (uint256)
           {
13         price = WBTC.balanceOf(Exchange) * WBTC.
               latestAnswer() / totalSupply();
14         return price;
15     }
16 }
```

(a) Contract `cWBTC` for token borrowing and redeeming.

```
1  contract Exchange{
2      function swap(uint amount0Out, uint amount1Out,
           address to) external {
3          if (amount0Out > 0) token0.transfer(to,
               amount0Out);
4          if (amount1Out > 0) token1.transfer(to,
               amount1Out);
5          ...
6      }
7  }
```

(b) Contract `Exchange` for token exchanging.

Figure 2: An example of SI bug in DApp `Inverse Finance`.

identify this SI bug based on the state changes of variable `price`. In other words, to what extent, the state changes should be considered an SI bug? Moreover, since the two functions (i.e., `borrowAll()` and `swap()`) do not have any explicit dependencies such as control flow and data flow, it is rather difficult for fuzzing-based approaches to precisely construct a feasible path (i.e., two critical functions are invoked by the same caller with a specific order) to expose such State Inconsistency bugs.

Our Solution. As mentioned earlier, in addition to emulating smart contract executions with on-chain state information to get precise contextual semantics, our approach integrates transaction fuzzing and differential analysis to identify SI bugs without specific testing oracles. More specifically, as it is non-trivial to confirm whether a state inconsistency bug exists based on testing oracles, it is much easier to check whether a transaction (e.g., borrowing assets) can be interfered with by any other look-innocent (but malicious) transactions in an unexpected way. Turning to this example, *IcyChecker* firstly records and replays the original transaction (i.e., `borrowAll()`) based on this historical information recorded on the blockchain. In this way, it can accurately recover the exact contextual information of the transaction. Then, *IcyChecker* associates the `swap()` function as relevant, as the contract address of this method is accessed by cWBTC (i.e., line 13 in Figure 2(a)). Then, *IcyChecker* performs transaction sequence mutation by reordering the two transactions that invoke these two functions (i.e., `swap()` and `borrowAll()`), and further compares the final transaction states between the original transaction sequence and the emulated one. In this way, *IcyChecker* identified that the contract state (i.e., attacker's balance of `WBTC`) will be changed due to the transaction order difference. Finally, it reports the transaction invoking `borrowAll()` is vulnerable to an SI bug, in which the attacker can manipulate the token price by invoking `swap()` before `borrowAll()`.

3 DESIGN OF *ICYCHECKER*

In this section, we first summarize the key patterns of SI bugs and their consequences (i.e., the specific vulnerabilities they may cause). Then we present the overall design of *IcyChecker*.

3.1 SI Patterns and Consequences

As mentioned earlier, a large portion of smart contract vulnerabilities (e.g., front-running, re-entrancy) is actually caused by SI bugs. Particularly, these vulnerabilities are actually due to the *hazardous access of smart contract states* [1]. To this end, we have summarized three typical scenarios that can lead to SI bugs and their connections to different vulnerabilities. This information is important for understanding how *IcyChecker* effectively recovers these scenarios via runtime fuzzing and exposes the potential SI bugs.

(1) In-batch transaction dependency. Since the order of submitted transactions can be manipulated by malicious miners [24], ignoring the possible state changes caused by other transactions in the same batch can also raise security issues. The most typical example of this SI pattern is the Front-running attack [1], in which an adversary initiates a malicious transaction before the target (victim) transaction. In this way, the attacker can alter the states of the targeted transaction and influence the execution results in the same batch. Besides, ignoring in-batch transaction dependency may also cause access control vulnerabilities in smart contracts, if the access control logic (e.g., a condition check) depends on the state information of an external contract.

(2) Control flow hijacking. Security-savvy smart contract developers may only consider cross-contract invocation under legitimate scenarios. However, this assumption may not always hold, as an adversary can easily hijack the control flow and update security-critical states in the middle of the victim transaction. This situation often happens in re-entrancy attacks, where the actual execution flow of re-entered contracts depends on state variables to be updated of the caller contracts [17]. For example, a defective caller contract of a DApp plays a role in borrowing an ERC-777 token, which transfers tokens before updating users' borrow amounts. During transferring of the token, the adversary can intentionally hijack the control flow, following the ERC-777 token specification. Additionally, there is another contract realizing access control policies of the borrowing market. By re-entering the access control contract in the hijacked control flow, the adversary can unexpectedly exit the borrowing market before the borrow amounts are updated, which is a state variable in the caller contract. Thus, the adversary obtains a non-repayable loan as he already exits the borrowing market.

(3) Block information dependency. Lastly, using block information to determine critical operations can lead to unexpected results.

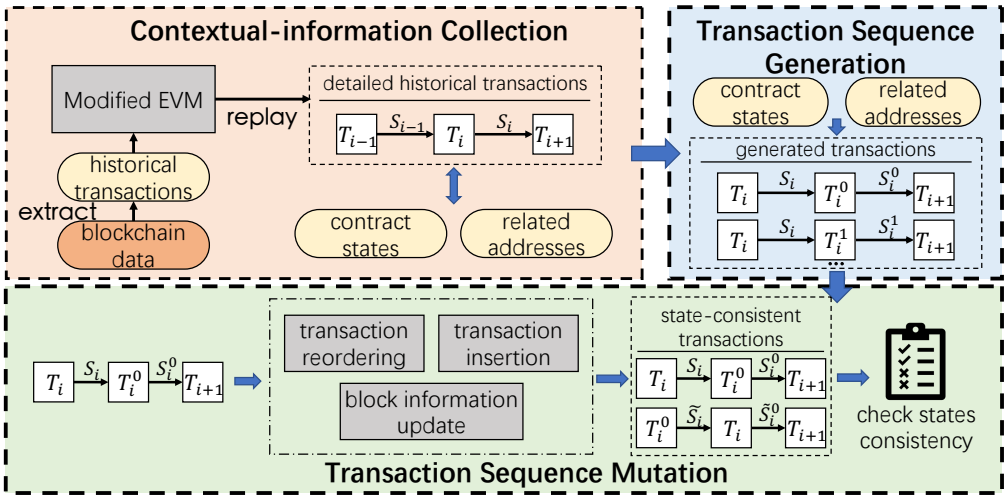


Figure 3: Overview of *IcyChecker*. In each transaction sequence, T_i in white squares refer to transactions or function calls, while S_i on the directed arrows refer to states after transaction T_i.

If a smart contract relies on block information (block states) to determine its critical logic, it is possible to include SI bugs as the block information can be changed by block miners to a certain extent. This scenario is particularly common in smart contracts for gaming and gambling, as they may take the block timestamp as a random seed to determine the game-winner. This type of vulnerability is also known as the Bad Randomness vulnerability [15].

As we will further show in Section 4, based on the above-explained key patterns of SI bugs, our approach generates a set of SI testing scenarios with different transaction sequences, and further confirm whether an interested transaction (or smart contract call) is indeed vulnerable.

3.2 Workflow of *IcyChecker*

Figure 3 shows the overview of *IcyChecker*. It mainly consists of three individual modules, namely, (1) Contextual-information collection (CIC), (2) Transaction sequence generation (TSG), and (3) Transaction sequence mutation (TSM).

In CIC, *IcyChecker* collects a set of critical information based on the historical transactions recorded on the blockchain. In order to obtain more detailed contextual information of these transactions, *IcyChecker* replays each transaction and records those related contract states before and after the transaction execution. In addition, by analyzing such historical transactions, *IcyChecker* identifies a set of related smart contracts based on the addresses associated with such historical transactions. These related contracts are more likely to have implicit dependencies with the DApp, as they may access the smart contract states via cross-contract communication.

In TSG, *IcyChecker* generates a set of feasible transaction sequences as the fuzzing candidates from the previously collected information. Here, each fuzzing sequence consists of two function calls: the original historical transaction (i.e., T_i) and the public methods (interfaces) of other related smart contracts (i.e., T_i^0 and T_i^1). In other words, in each sequence, *IcyChecker* extends the original historical transaction by adding public methods of other related smart contracts based on the associated addresses. In the meantime, to better emulate the actual contextual information as the on-chain data, such transaction sequences are executed with the on-chain state information before the historical transaction.

In TSM, *IcyChecker* implements a set of mutation strategies to generate *state-consistent transaction sequences* from the fuzzing candidates in the previous step. To inspect whether any transaction interactions may cause SI bugs as we summarized in Section 3.1, *IcyChecker* mutates transaction sequences by mechanisms such as transaction reordering, insertion, and block information update. By executing the mutated transaction sequences and observing the interested final states (i.e., S_i^0 and $\tilde{S}_i^0$), *IcyChecker* identifies whether an SI bug exists with the given transaction sequences.

4 APPROACH DETAILS

4.1 Contextual-information Collection

IcyChecker first analyzes and obtains the details of historical transactions of a given DApp from the raw Ethereum data, including the before and after states of each transaction, corresponding block information, as well as related addresses which are associated with these transactions.

Transaction detail collection. By requesting peers of the Ethereum mainnet, we obtain transaction history in each block, as well as block information such as block number, timestamp, and difficulty.

Since blockchain only records the latest states of the smart contract, to obtain the historical states of each transaction, *IcyChecker* replays each transaction offline by starting from the genesis block (initial block) of each DApp. Here, *IcyChecker* builds up a replay infrastructure on top of an off-the-chain execution platform [23], which is specifically designed for smart contract analysis. *IcyChecker* instruments the customized EVM and accesses the concrete state variables before and after the execution of each history transaction. We record the state variables before each transaction as a three-tuple, namely, < `addr`, < `key`, `value` >>. Given a pre-state

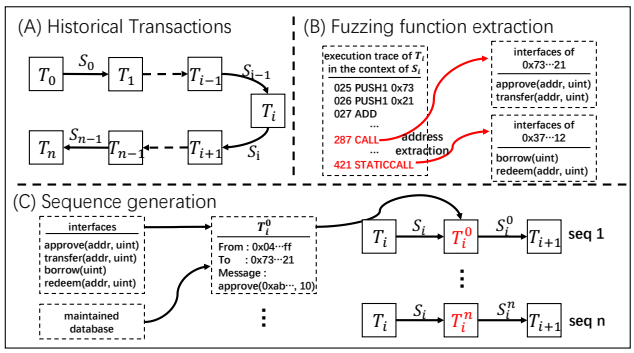


Figure 4: Detailed process of TSG.

$<$ addr, $<$ key_0, value_0 $>>$ as well as a transaction, the post-state will be collected as $<$ addr, $<$ key_1, value_1 $>>$ if corresponding states (i.e., value) are modified by the transaction.

Related address collection. In addition to historical transactions, *IcyChecker* also extracts the addresses that appeared in the historical transactions. There are generally two types of addresses included in each transaction - the address of externally owned accounts (i.e., EOA address) [48], and the address of other smart contracts (i.e., contract address). The EOAs are wallet accounts controlled by DApp users, whereas the contract addresses are smart contracts that are independent, executable code deployed on the Ethereum network. As we will further show in Section 4.2, these addresses are further used to more effectively generate and execute valid transaction sequences for detecting SI bugs.

To this end, by analyzing the bytecode in EVM, *IcyChecker* extracts addresses associated with opcode such as `CALL`, `CALLCODE`, `DELEGATECALL`, `STATICCALL`, as these opcodes are responsible for function invocations. In addition, to more thoroughly collect the potential relevant EOAs and smart contracts, we also collect the nested addresses within the associated smart contracts. Note that due to the gas usage limitation in Ethereum, such an address collection process typically ends within a few cross-contract calls from the starting contract.

Example. Assuming a sender invokes the *transfer* function of an ERC-721 token, in which the ownership of this token would be transferred from the sender to the recipient. In this case, the state variable representing the ownership of this token in this contract would be modified accordingly. In the meantime, by analyzing the EVM bytecode of function transfer, *IcyChecker* finds it invokes a callback function of the recipient through the `CALL` opcode. Therefore, it also labeled the recipient address as a relevant smart contract for future fuzzing.

4.2 Transaction Sequence Generation

Fuzzing functions extraction. In order to trigger the potential SI bugs, *IcyChecker* collects functions (e.g., `approve()` and `transfer()`) that are capable to change contract state of the DApp. While the contract state may be changed by any external function call, to narrow down the exploration scope in the fuzzing process, *IcyChecker* generates transaction sequences by including other function calls (as fuzzing functions) in its associated smart contracts based on the previously identified contract addresses, as these functions are more likely to cause SI bugs.

To this end, for each identified smart contract (e.g., contract 0x73...21 and contract 0x37...12 in Figure 4(B)) that appeared in the historical transaction (i.e., T_i) for fuzzing, *IcyChecker* extracts its available public interfaces based on the ABI information of the contract. Note that in this process, *IcyChecker* excludes functions with *view* and *pure* types, as they cannot modify any state variables. In the meantime, the parameter types of each function call are also collected for triggering a valid function invocation.

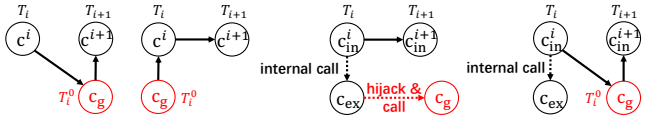
Sequence generation. With the collected external function calls (e.g., `approve()` in Figure 4(C)) and a specific historical transaction (i.e., T_i), *IcyChecker* constructs a set of transaction sequences for future fuzzing (e.g., `seq 1` and `seq n`). More specifically, for each historical transaction in the DApp, *IcyChecker* associates one related function and forms a new transaction sequence each time. For example, given one historical transaction and ten external function calls identified from another smart contract, *IcyChecker* generates ten unique transaction sequences.

Fuzzing context (input) preparation. In addition to performing fuzzing with the actual states of historical transactions, *IcyChecker* takes the following additional mechanisms to obtain the fuzzing input (i.e., function parameters and callers) for transaction sequence execution. Compare to randomly generating these inputs as in previous fuzzing approaches [21], these mechanisms provide better-quality inputs that are closer to real-world contexts.

- **Function parameters.** *IcyChecker* maintains a specific data structure to record the concrete variable values during transaction replay (in Section 4.1). Each unique parameter value is recorded with its parameter type, as well as the latest timestamp. In the runtime-fuzzing process, *IcyChecker* provides the concrete parameter value by querying the database and using the one whose timestamp is the closest to the corresponding historical transaction. A random parameter value is generated unless there is no such concrete value of the same type in the historical data.
- **Function (contract) callers.** Similar to constructing function parameters, *IcyChecker* maintains three types of External Owned Accounts (EOAs) as the possible function caller (i.e., `From` in T_i^0) when executing the generated sequence. These EOAs include (1) addresses related to historical transactions collected previously (see Section 4.1), (2) other participants of the DApp, and (3) randomly generated EOAs. During this selection procedure, we make historical transaction-related addresses the highest priority, while the randomly generated EOAs as the lowest.

4.3 Sequence Mutation with Differential Analysis

In this part, *IcyChecker* mutates the previously generated sequences and generates new transaction sequences. The original sequences with each of its mutated sequences are treated as a *sequence pair*, in which the final states of DApp after executing the two sequences should be the same (i.e., with consistent final states). More specifically, following the SI patterns as mentioned in Section 3.1, for



(a) transaction reordering. **(b) transaction insertion.**

Figure 5: Proposed mutation strategies in *IcyChecker*.

each historical transaction (and its generated fuzzing sequence) in the DApp, the mutation strategies aim at inspecting the following scenarios to find SI bugs:

- With transactions from distinct callers in the same batch, the final DApp states should be the same regardless of the order of these transactions.
- By adding an arbitrary transaction call that hijacks the control flow of the historical transaction, the final DApp states should be equivalent to adding that extra transaction after finishing executing the historical transaction.
- The execution result of transactions should be the same when block information such as block timestamp and difficulty is changed within a reasonable margin.

Otherwise, the DApp and the specific historical transaction to generate the is considered to have an SI bug.

Sequence mutation strategies. As mentioned in Section 3.1, there are three possible scenarios that may lead to SI bugs, namely, in-batch transaction dependency, control flow hijacking, and block information dependency. Therefore, for each historical transaction in the DApp, *IcyChecker* implements the following three strategies for sequence mutation.

(1) Transaction reordering. As shown in Figure 5(a), to detect in-batch transaction dependency, *IcyChecker* reverses the order of the two transactions (i.e., the historical transaction T_i calling contract c^i and the external contract call T_i^0 calling contract c_g) in the generated transaction sequence, and execute transactions under the same batch. If the two transactions from distinct callers do not have a state dependency relation, the final DApp states of the original and mutated transaction sequence should be exactly the same.

(2) Transaction insertion. To find out SI bugs caused by control flow hijacking, *IcyChecker* intentionally inserts the external function call within the historical transaction. More specifically, as shown in Figure 5(b), in the original sequence, the external call (i.e., T_i^0) is executed after the historical transaction (i.e., T_i). To mutate this transaction, *IcyChecker* inspects the bytecode of the historical transaction and finds out its internal call (e.g., calling c_{ex}) based on call instructions and the corresponding address. Then, *IcyChecker* inserts the external call in c_{ex}, a contract that does not belong to the DApp. In this way, the external call successfully hijacks the original control flow of the original sequence.

(3) Block information update. Lastly, *IcyChecker* executes the same transaction with different block information such as block timestamp and block difficulty, as the block timestamp can be manipulated by miners and block difficulty can also be changed by Ethereum updates.

With the above three mutation strategies, *IcyChecker* generates transaction pairs, executes these transactions with the same contextual information, and checks whether the final states of the DApps remain the same. If there are any inconsistent DApp states among the two transaction sequence pair, we consider the DApp vulnerable to an SI bug. In addition, this transaction sequence pair actually provides the exact exploit for debugging and diagnostics.

5 IMPLEMENTATION

We implement *IcyChecker* with around 10,000 lines of code in Go language. To record and replay historical transactions, we build *IcyChecker* on top of an off-chain execution platform [23] which supports fast re-execution of individual transactions for program analysis and testing. By simulating the distributed blockchain system, the platform can accurately recover and record the original states used by each historical transaction (i.e., transaction-relevant states) of a DApp.

Historical data collection. In addition to recording all states of historical transactions, in the CIC module (in Section 4.1), *IcyChecker* collects and stores other contextual information such as related contract addresses, concrete function parameter values, which are further used for transaction sequences execution (in Section 4.2). Note that if analyzing each DApp individually, it may need to repeatedly replay historical transactions of the same token contract multiple times. To improve the efficiency of CIC, *IcyChecker* replays historical transactions of all DApps together and records the corresponding state changes only once.

To extract external contract calls for fuzzing, the Transaction Sequence Generation (TSG) component has an interpreter for parsing ABI files. The interpreter analyzes ABI files of targeted contracts, reads function signatures as well as their parameter types, and finally stores such information in JSON format in the database. Specifically, each function call is transformed into a hexadecimal form with a 32-bit length. Each parameter of the function occupies a 256-bit space which is the maximum size of any possible parameter value. Each parameter of the function occupies one or multiple slots of the 256-bit space.

Sequence execution. Similar to other transaction-based analysis frameworks (e.g., sFuzz [33]), *IcyChecker* implements a customized Ethereum Virtual Machine (EVM) [48] which supports runtime instrumentation to execute the generated and mutated transaction sequences. During the sequence execution, *IcyChecker* constantly checks the parameter values of the same type from its historical data (i.e., transaction replay), and uses the corresponding value to invoke contract calls. If there is no suitable data in the persistent storage, *IcyChecker* randomly generates the concrete input based on its required size.

To perform differential analysis over the original and mutated sequences, *IcyChecker* creates two EVM instances and duplicates the necessary state variables. Each EVM instance executes an individual sequence and the final DApp states in the two EVM instances are compared to identify SI bugs.

6 EVALUATION

To evaluate the effectiveness of *IcyChecker*, we inspect the latest 1,000 historical transactions (as of Apr. 2022) of the most popular

DApps. We analyze the reported results via manual inspection. In addition, we compare *IcyChecker* with three state-of-the-art tools, i.e., Smartian [7], Confuzzius [44], and Sailfish [1]. Note that to avoid inaccurate evaluation results caused by randomness, we repeat each experiment five times when running *IcyChecker*. Lastly, we report the performance overhead of *IcyChecker*.

6.1 Evaluation Setup

To evaluate the effectiveness of *IcyChecker*, we analyze the top 100 DApps from DAppRadar [11], one of the most popular DApp stores. More specifically, we collect the contract addresses of each DApp, as well as the collection of ABI information of these DApps. For each DApp, any contract address out of its address collection is considered an external contract.

To obtain the historical transaction (contract) states, we collect all the raw Ethereum block data as of Apr-01-2022, which is the 14,500,000-th block. For transaction fuzzing, we use the latest 1,000 historical transactions as the fuzzing seeds for sequence construction and mutation. As a fuzzing-based approach, we consider the recent 1,000 transactions are sufficient to cover most DApp usage scenarios.

Table 1: Environmental setup used by *IcyChecker*.

Server	CIC	TSG & TSM
CPU	Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz × 2, 40 cores × 2 sockets	
RAM	512GB DDR4 RAM @ 2133MHz	
SSDs	Intel Optane P4800X @ 1.5TB, Intel SSD P5510 @ 7.68TB	Samsung SSD 980 PRO @ 2TB
OS	Ubuntu 20.04.1 LTS	

All experiments in our evaluation are conducted on a machine with two Intel(R) Xeon(R) Gold 5218R CPUs @ 2.10GHz, 512GB RAM, and Ubuntu 20.04.1 OS. Table 1 presents other key environmental setups which are used in our evaluation.

6.2 Effectiveness

Table 2 shows the detailed results of *IcyChecker* by analyzing the DApps and their recent 1,000 historical transactions. As can be seen from the Table, *IcyChecker* totally reports 277 state inconsistent instances of the tested DApps in five runs. Note that, since the same SI bug can be triggered multiple times under different fuzzing inputs, we remove the duplicated results which are reported by the same transaction sequence (in Section 4.2).

Precision. To obtain the final precision of *IcyChecker*, we manually inspect each of the reported SI cases and justify their correctness. Our analysis showed that *IcyChecker* achieves an overall precision of 87% (241/277) in detecting the SI bugs. The accuracy of *IcyChecker* in detecting SI bugs caused by block information dependency reaches 100%. Additionally, 45 of identified bugs can lead to vulnerabilities. The rest of identified bugs are related to fairness issues that impact DApp users, which is also recognized by previous research [13]. For example, a DApp owner can arbitrarily cancel users' bids in an auction.

False positives. During our manual inspection, we find that the false positives caused by *IcyChecker* are mainly due to the following two reasons. (1) In some cases, *IcyChecker* mistakenly treats an internal contract address as an external address, as some of the DApp addresses are not thoroughly collected from the DApp store (e.g., the addresses of logic contracts in proxy patterns). In this way, *IcyChecker* may report false SI bugs which are related to re-entrancy vulnerabilities. (2) While *IcyChecker* extracts as much contextual information as possible for fuzzing, false positives may take place if no such appropriate information is provided. For example, for some DApps, *IcyChecker* may only find one valid function callee for function fuzzing, and hence report false alarms of SI bugs.

Table 2: Overall effectiveness of *IcyChecker* in detecting different SI patterns.

State-inconsistency pattern	#Unique instances reported by *IcyChecker*	Manual inspection		
		#TP	#FP	Precision
In-batch transaction dependency	139	112	27	81%
Control flow hijacking	23	14	9	61%
Block information dependency	81	81	0	100%
Average	243	207	36	85%
Total (in five runs)	277	241	36	87%

False negatives. Due to the lack of ground truth, we are not able to evaluate the false negatives (i.e., missed SI bugs) in *IcyChecker*. However, as we will further show in Section 6.4, by comparing *IcyChecker* with other state-of-the-art tools over the same set of DApps, *IcyChecker* only misses 11 SI bugs detected by other tools. The false negatives are mainly caused due to the lack of appropriate contextual information (e.g., an external transaction invoking the vulnerable functions) during transaction fuzzing.

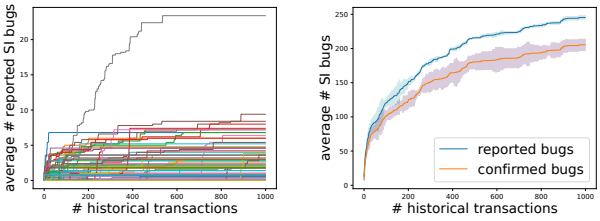
Effectiveness of accurate context information. One of the key designs in *IcyChecker* is it utilizes more accurate context information to execute the generated transaction sequences (i.e., historical transaction states, valid parameter values, and related external calls). To this end, we manually inspect ten reported SI bugs and check if such bugs can be triggered by a purely random strategy.

Not surprisingly, among the reported SI bugs, three of them can only be triggered under specific parameters (e.g., a given address), while two of them need to be triggered with a specific caller (e.g., a valid DApp user). Figure 6 shows an example of these cases, where malicious users can steal extra money by front-running `approve()` with `transferFrom()`. As can be seen, this bug can only be triggered when the input of `src` equals to `msg.sender` of `approve()`. In other words, it is nearly impossible to generate such a valid input randomly.

```
function approve(address spender, uint256 amount) {
    transferAllowances[msg.sender][spender] = amount;
}

function transferFrom(address src, uint256 amount) {
    transferAllowances[src][msg.sender] -= amount;
    ...
}
```

Figure 6: An example of SI bug which requires specific contextual information to trigger.



(a) The average number of reported SI bugs in each DApp.

(b) The average number of reported/confirmed SI bugs among five runs.

Figure 7: The average number of identified SI bugs based on the number of historical transactions for fuzzing.

Effectiveness of fuzzing strategies. In addition, thanks to the unique fuzzing strategies in *IcyChecker*, it can effectively discover most of the potential SI bugs by fuzzing a very limited number of historical transactions (e.g., only hundreds of transactions).

Figure 7(a) shows the average number of triggered alarms of each DApp corresponding to the number of historical transactions. Each DApp is distinguished with distinct colors. As can be seen, most alarms are triggered within the first 400 transactions. Note that the number of SI bugs identified in DApp *aave* is much more than others (i.e., the grey line on the upper side in Figure 7(a)). This is because many of the SI bugs detected in this app implement the same bytecode in distinct scenarios (in different smart contracts). Additionally, Figure 7(b) shows the average number of triggered alarms and confirmed bugs of multiple runs with the corresponding standard deviation. Since the growth trend of identified bugs is a logarithmic curve, it indicates that *IcyChecker* can trigger SI bugs within a relatively smaller number of transaction sequences.

6.3 Performance Overhead

The performance overhead of *IcyChecker* lies in two parts: (1) transaction record and replay for contextual information collection, and (2) transaction generation, mutation, and execution. Firstly, with the experimental setup as shown in Section 6.1, it takes *IcyChecker* around 300 hours and 4 TB of storage space to replay and record all transactions from the genesis block to the 14,500,000-th block. Note that, while this process is relatively costly, the process of transaction record and replay is a one-time effort. In addition, the overhead can be reduced with more powerful computing resources. Secondly, for each DApp, it takes around 60 minutes on average to process our fuzzing strategies over the 1,000 historical transactions. Given the effectiveness of *IcyChecker* in terms of reporting SI bugs, we consider such an overhead acceptable.

6.4 Comparison with other Tools

As mentioned earlier, to better justify the effectiveness of *IcyChecker*, we compare it with the three most representative state-of-the-art tools, namely, Smartian [7], Confuzzius [44], and Sailfish [1]. Particularly, Sailfish is a static analyzer specifically designed for detecting SI bugs via symbolic analysis. Smartian and Confuzzius are dynamic fuzzers, which take bytecode and ABI as input. We obtain the publicly released artifacts of the three frameworks from their corresponding publications. Further, we run the three frameworks over the same 100 DApps in comparison with *IcyChecker* and manually identify false alarms of these comparison tools. Note that we did not compare *IcyChecker* with other popular tools such as Securify [45], Mythril [9], sFuzz [33], as they have been proven less effective than Sailfish in terms of effectiveness and efficiency [1].

Table 3: Comparing *IcyChecker* with Smartian, Confuzzius, and Sailfish.

Bug Type	Smartian		Confuzzius		Sailfish		*IcyChecker*	
	#Total	#TP	#Total	#TP	#Total	#TP	#Total	#TP
Front-running	N/A	N/A	1	1	7	2	139	112
Re-entrancy	7	2	1	0	10	1	23	14
Bad Randomness	0	0	13	12	N/A	N/A	81	81
Average	7	2	15	13	17	3	243	207
Total (in five runs)	8	2	17	13	17	3	277	241

Table 3 shows the experiment result, where average and total reported bugs as well as confirmed bugs in five runs of the four tools are listed. Here, the cells with `N/A` mean the tool cannot detect the corresponding SI bug. As can be seen from the table, *IcyChecker* successfully identifies 239, 228, and 238 more SI bugs than Smartian, Confuzzius, and Sailfish, respectively. The false positive rate of *IcyChecker* is much lower than the other tools (with only 13% FP rate, compared to 75% in Smartian, 24% in Confuzzius, and 82% in Sailfish).

Like most smart contract fuzzers (e.g., ILF [18], sFuzz [33] and ContractFuzzer [21]), Smartian and Confuzzius are designed to deploy contracts on a private chain for fuzzing. Due to the lack of accurate contextual information as collected by *IcyChecker* through transaction replay, Smartian and Confuzzius suffer from a really low success rate. Similarly, as a static analysis approach, Sailfish suffers from an even higher false positive rate due to the lack of considering the context of state variables. For example, without considering the exact value of a condition variable, Sailfish may mistakenly report a re-entrancy vulnerability which is not feasible in practice.

In terms of false negatives, our manual inspection revealed that *IcyChecker* had a low false negative, detecting all but 11 true SI bugs reported by other tools. Again, as mentioned earlier, while the other three frameworks successfully identify such SI bugs, their approaches introduce a significant number of false positives as the trade-off.

To this end, we conclude that *IcyChecker* indeed performs better than the other three frameworks in terms of detecting SI bugs.

7 DISCUSSION AND LIMITATION

In this section, we first give a detailed case study to show how *IcyChecker* benefits from its key designs and hence reports unique SI bugs with new attack patterns. Then, we discuss the limitations of *IcyChecker* and the potential future works.

7.1 Case Study - Design Flaw of DApp `Compound`

The DApp `Compound` is a financial application, where users can deposit and earn interests via its governance token `Comp`. A user will

```
contract Reservoir {
    /* Drips the maximum amount of tokens to match the
         drip rate since inception */
  function drip() public returns (uint) {
    ...
    uint reservoirBal = token.balanceOf(address(this));
    // Next, calculate intermediate values
    uint dripTotal = mul(dripRate, blockNumber -
         dripStart, "dripTotal overflow");
    uint deltaDrip = sub(dripTotal, dripped, "deltaDrip
         underflow");
    uint toDrip = min(reservoirBal, deltaDrip);
    // Finally, transfer tokens to target
    token.transfer(target, toDrip);
    return toDrip;
  }
}
```

(a) contract `Reservoir`.

```
contract Unitroller{
    /* Transfer COMP to the user */
    function grantCompInternal(address user, uint amount
         ) internal returns (uint) {
        Comp comp = Comp(getCompAddress());
        uint compRemaining = comp.balanceOf(address(this
             ));
        if (amount > 0 && amount <= compRemaining) {
            comp.transfer(user, amount);
            return 0;
        }
        return amount;
    }
}
```

(b) contract `Unitroller`.

Figure 8: DApp `Compound`.

receive more shares of `Comp` tokens with a higher deposit amount. With this token, participants can create and vote for governance proposals. As shown in Figure 8(a), in this DApp, the function `drip()` transfers `Comp` token at a certain rate to the `Unitroller` contract. The function `grantCompInternal()` in Figure 8(b) is utilized to claim rewards by participants[2]. Due to the lack of proper access control of these two critical functions, a considerable amount of `Comp` tokens could get lost as the attacker can easily set up an incorrect distribution rate. More specifically, the `drip()` function can be simultaneously called by distinct users, leading to unnecessary loss to DApp users; In other words, an adversary can gain additional profits by setting an incorrect distribution rate, while users may suffer from unfair losses due to missing triggering the `drip()` function.

Existing methods fail to detect this design flaw due to the lack of accurate contextual information and effective oracles. In this case, app users can claim a certain amount of `Comp` based on the corresponding balance of the `Unitroller` contract. Additionally, the `Reservoir` contract can manipulate the state of balance in the same batch of transactions. For static analysis approaches that inspect contract code, both of the two contracts work well when considered individually unless the `drip()` function is executed in the same batch. Due to the lack of adequate cross-contract analysis with such contextual information, this case is overlooked. In the meantime, dynamic analysis tools require a specific testing oracle to catch the abnormal balance change (e.g., a state variable), which is actually not feasible under the scenario of offline simulation.

The differential analysis eases the problem of oracles. By generating transactions invoking these two functions and executing them in different orders, the final user balance will be obviously different. To this end, it is clear to *IcyChecker* that this case leads to a state inconsistency bug. To identify this SI bug, the balance difference here can only be triggered with *non-zero balance of users* and the distributing contract. In addition, the state variable should be properly conserved so as to run the whole DApp. *IcyChecker* combines real contract states with effective fuzzing mechanisms to locate this bug, instead of spawning substantive transactions and randomly generating proper states as in previous approaches.

[2]Due to space limitation, details about how such an internal function is invoked by users are omitted

7.2 Limitations and future works

Admittedly, *IcyChecker* has the following limitations which can be further improved by future works. Firstly, as mentioned earlier in Section 6.2, *IcyChecker* incurs relatively high overhead for obtaining accurate context information during transaction record and replay. Under the same experimental settings, the overhead will also get increased, as the number of blocks on Ethereum keeps increasing. Therefore, the performance overhead of *IcyChecker* can be reduced by more advanced off-the-chain execution infrastructure. Secondly, during the sequence fuzzing sequence execution (i.e., in Section 4.2), *IcyChecker* mainly chooses the latest concrete parameter values as the fuzzing input. To identify more SI bugs, more in-depth analysis can be performed to select other high-quality inputs from the collected contextual information. Finally, in our research, we only explored the top 100 DApps with their top 1,000 latest transactions. To identify more SI bugs, it is necessary to adopt *IcyChecker* with more DApps and historical transitions.

8 RELATED WORK

This work is closely related to various approaches for detecting smart contract vulnerabilities under different settings.

Static analysis. Great efforts have been paid to identify smart contract vulnerabilities by analyzing the source code [22, 26, 27, 38, 43] or decompiled bytecode [2, 14, 16, 45] of smart contracts through static analysis. For example, Feist et al. propose a static analyzer, Slither, which can effectively detect major smart contract vulnerabilities like re-entrancy, front-running, and integer error. Due to the increasing complexity of smart contract code and vulnerability patterns, more recent works start detecting cross-contract vulnerabilities by integrating more in-depth data flow and control flow analysis [25, 29, 50, 52]. Particularly, as mentioned earlier in Section 2.3, SailFish [1] is specifically designed for detecting SI bugs via static analysis. However, due to the lack of runtime contextual information, SailFish shares inherent limitations as in other works [4, 28, 29] caused by static analysis.

In the meantime, a lot of researches identify smart contract vulnerabilities based on the real-world transactions of smart contracts [5, 10, 34, 35, 40, 53]. For example, Perez et al. [34] tracked the

execution procedure of transactions to identify whether vulnerabilities have been exploited. Some other research [5, 10, 35] identified unusual behaviors of contracts by analyzing transactions, including inconsistency with regard to ERC-20 standards [5], and token price manipulation [35]. However, most of these works are based on specific heuristics for vulnerability detection, making them less generic to new variants of exploits and attacks.

Dynamic analysis. In addition to statically analyzing contract code and transactions, various dynamic fuzzing approaches [7, 18, 21, 33, 51] are also proposed for detecting smart contract vulnerability. With more accurate semantic information obtained via runtime execution, dynamic analysis significantly helps to eliminate those false alarms caused by static analysis. Following this line of research, recent works propose fine-grained fuzzing mechanisms to refine the testing oracles and improve path coverage for vulnerability detection. For example, Choi et al. [7] collected data flow information by static and dynamic analysis and further feed them to pattern-based test oracles in their fuzzing process. He et al. [18] improve the path coverage for intra-contract analysis by generating more effective transaction sequences. xFuzz [51] integrates machine learning to guide the fuzzer to identify more cross-contract scenario vulnerabilities. Different from these works, our research is specifically designed for detecting SI bugs by fuzzing valid transactions. With the three SI patterns identified in our approach, *IcyChecker* does not rely on a specific testing oracle, hence making it generic to locate various unknown SI bugs.

The mutation strategies in *IcyChecker* (in Section 4.3) are similar to the concept of Metamorphic testing [6]. Metamorphic testing has been widely used in various software testing scenarios other than smart contracts, such as RESTful Web APIs [37], Simulation Program [20], Datalog Engines [30], artificial intelligence [19, 42], NLP [36], Android Apps [41]. For example, Su et al. propose a metamorphic testing-based approach to detect non-crashing logic bugs [41]. To the best of our knowledge, our research is the first of its kind to adopt metamorphic testing for detecting state inconsistency bugs in smart contracts. We believe more metamorphic relations can be proposed to detect other bugs and vulnerabilities in smart contracts.

9 CONCLUSION

In this paper, we present *IcyChecker*, a new transaction-based fuzzing framework for detecting state inconsistency bugs in smart contracts (DApps). Particularly, *IcyChecker* extracts the on-chain states and other related information (e.g., external function calls and parameter values) by replaying the historical transactions. In this way, this information provides accurate contextual information for fuzzing smart contracts of the DApps. In addition, *IcyChecker* implements a set of novel sequence mutation strategies to identify SI bugs without relying on specific testing oracles. We conduct experiments to show that *IcyChecker* outperforms state-of-the-art tools in terms of finding real-world SI bugs.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their detailed and valuable comments. This work was supported in part by the National Key R&D Program of China (No.2020YFB1707603), the National Natural Science Foundation of China (No.62032025, No.62202510), the Special Projects in Key Fields of Universities in Guangdong Province (No.2022ZDZX1001), the Fundamental Research Funds for the Central Universities, Sun Yat-sen University (No.22lgqb26), and the WeBankScholars Program.

REFERENCES

[1] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. 2022. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 161–178.
[2] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. 2020. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 454–469.
[3] Wei Cai, Zehua Wang, Jason B Ernst, Zhen Hong, Chen Feng, and Victor CM Leung. 2018. Decentralized applications: The blockchain-empowered software system. *IEEE Access* 6 (2018), 53019–53033.
[4] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2021. Defectchecker: Automated smart contract defect detection by analyzing evm bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2189–2207.
[5] Ting Chen, Yufei Zhang, Zihao Li, Xiapu Luo, Ting Wang, Rong Cao, Xiuzhuo Xiao, and Xiaosong Zhang. 2019. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in ethereum. In *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 1503–1520.
[6] Tsong Y Chen, Shing C Cheung, and Shiu Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543* (2020).
[7] 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 *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 227–239.
[8] CNVD. 2012. China National Vulnerability Database. https://www.cnvd.org.cn/
[9] ConsenSys. 2022. Mythril. https://github.com/ConsenSys/mythril
[10] Philip Daian, Steven Goldfeder, Tyler Kell, Yunqi Li, Xueyuan Zhao, Iddo Bentov, Lorenz Breidenbach, and Ari Juels. 2020. Flash boys 2.0: Frontrunning in decentralized exchanges, miner extractable value, and consensus instability. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 910–927.
[11] DappRadar. 2022. DappRadar - The World's Dapp Store | Blockchain Dapps Ranked. https://dappradar.com/
[12] DefiLlama. [n. d.]. DefiLlama - DeFi Dashboard. https://defillama.com/
[13] Yue Duan, Xin Zhao, Yu Pan, Shucheng Li, Minghao Li, Fengyuan Xu, and Mu Zhang. 2022. Towards Automated Safety Vetting of Smart Contracts in Decentralized Applications. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 921–935.
[14] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.
[15] João F Ferreira, Pedro Cruz, Thomas Durieux, and Rui Abreu. 2020. Smartbugs: A framework to analyze solidity smart contracts. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 1349–1352.
[16] Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. Madmax: Surviving out-of-gas conditions in ethereum smart contracts. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–27.
[17] Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2017. Online detection of effectively callback free objects with applications to smart contracts. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–28.
[18] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin 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*. 531–548.
[19] Pinjia He, Clara Meister, and Zhendong Su. 2020. Structure-invariant testing for machine translation. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 961–973.
[20] Xiao He, Xingwei Wang, Jia Shi, and Yi Liu. 2020. Testing high performance numerical simulation programs: experience, lessons learned, and open issues. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 502–515.
[21] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 259–269.
[22] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[23] Yeonsoo Kim, Seongho Jeong, Kamil Jezek, Bernd Burgstaller, and Bernhard Scholz. 2021. An Off-The-Chain Execution Environment for Scalable Testing and Profiling of Smart Contracts. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 565–579.

[24] 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*. 363–373.

[25] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 752–764.

[26] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. 2018. Reguard: finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*. IEEE, 65–68.

[27] Ye Liu, Yi Li, Shang-Wei Lin, and Rong Zhao. 2020. Towards automated verification of smart contract fairness. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 666–677.

[28] 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*. 254–269.

[29] Fuchen Ma, Zhenyang Xu, Meng Ren, Zijing Yin, Yuanliang Chen, Lei Qiao, Bin Gu, Huizhong Li, Yu Jiang, and Jiaguang Sun. 2021. Pluto: Exposing vulnerabilities in inter-contract scenarios. *IEEE Transactions on Software Engineering* 48, 11 (2021), 4380–4396.

[30] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic testing of datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 639–650.

[31] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M Kim, and Marek Laskowski. 2019. Understanding a revolutionary and flawed grand experiment in blockchain: the DAO attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32.

[32] Tian Min, Hanyi Wang, Yaoze Guo, and Wei Cai. 2019. Blockchain games: A survey. In *2019 IEEE conference on games (CoG)*. IEEE, 1–8.

[33] 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 ACM/IEEE 42nd International Conference on Software Engineering*. 778–788.

[34] Daniel Perez and Benjamin Livshits. 2021. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. 1325–1341.

[35] Kaihua Qin, Liyi Zhou, and Arthur Gervais. 2022. Quantifying blockchain extractable value: How dark is the forest?. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 198–214.

[36] Marco Tulio Ribeiro, Tongshuang Wu, Carlos Guestrin, and Sameer Singh. 2020. Beyond Accuracy: Behavioral Testing of NLP Models with CheckList. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4902–4912.

[37] Sergio Segura, José A Parejo, Javier Troya, and Antonio Ruiz-Cortés. 2018. Metamorphic testing of RESTful web APIs. In *Proceedings of the 40th International Conference on Software Engineering*. 882–882.

[38] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. 2021. {SmarTest}: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language {Model-Guided} Symbolic Execution. In *30th USENIX Security Symposium (USENIX Security 21)*. 1361–1378.

[39] Jianzhong Su, Hong-Ning Dai, Lingjun Zhao, Zibin Zheng, and Xiapu Luo. 2022. Effectively Generating Vulnerable Transaction Sequences in Smart Contracts with Reinforcement Learning-guided Fuzzing. In *37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[40] Liya Su, Xinyue Shen, Xiangyu Du, Xiaojing Liao, XiaoFeng Wang, Luyi Xing, and Baoxu Liu. 2021. Evil under the sun: understanding and discovering attacks on Ethereum decentralized applications. In *30th USENIX Security Symposium (USENIX Security 21)*. 1307–1324.

[41] Ting Su, Yichen Yan, Jue Wang, Jingling Sun, Yiheng Xiong, Geguang Pu, Ke Wang, and Zhendong Su. 2021. Fully automated functional fuzzing of Android apps for detecting non-crashing logic bugs. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–31.

[42] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. 303–314.

[43] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. Smartcheck: Static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain*. 9–16.

[44] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. 2021. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 103–119.

[45] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Buenzli, and Martin Vechev. 2018. Securify: Practical security analysis of smart contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 67–82.

[46] Qin Wang, Rujia Li, Qi Wang, and Shiping Chen. 2021. Non-fungible token (NFT): Overview, evaluation, opportunities and challenges. *arXiv preprint arXiv:2105.07447* (2021).

[47] Sam M Werner, Daniel Perez, Lewis Gudgeon, Ariah Klages-Mundt, Dominik Harz, and William J Knottenbelt. 2021. Sok: Decentralized finance (defi). *arXiv preprint arXiv:2101.08778* (2021).

[48] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32.

[49] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. 2021. Defiranger: Detecting price manipulation attacks on defi applications. *arXiv preprint arXiv:2104.15068* (2021).

[50] Yinxing Xue, Mingliang Ma, Yun Lin, Yulei Sui, Jiaming Ye, and Tianyong Peng. 2020. Cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1029–1040.

[51] Yinxing Xue, Jiaming Ye, Wei Zhang, Jun Sun, Lei Ma, Haijun Wang, and Jianjun Zhao. 2022. xFuzz: Machine Learning Guided Cross-Contract Fuzzing. *IEEE Transactions on Dependable and Secure Computing* (2022).

[52] Jiaming Ye, Mingliang Ma, Yun Lin, Yulei Sui, and Yinxing Xue. 2020. Clairvoyance: cross-contract static analysis for detecting practical reentrancy vulnerabilities in smart contracts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 274–275.

[53] Shunfan Zhou, Malte Möser, Zhemin Yang, Ben Adida, Thorsten Holz, Jie Xiang, Steven Goldfeder, Yinzhi Cao, Martin Plattner, Xiaojun Qin, et al. 2020. An ever-evolving game: Evaluation of real-world attacks and defenses in ethereum ecosystem. In *29th USENIX Security Symposium (USENIX Security 20)*. 2793–2810.

Received 2023-02-16; accepted 2023-05-03