FairChecker: Detecting Fund-stealing Bugs in DeFi Protocols via Fairness Validation

Yi Sun
Purdue University
West Lafayette, USA
sun624@purdue.edu

Zhuo Zhang
Purdue University
West Lafayette, USA
zhan3299@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, USA
xyzhang@cs.purdue.edu

***Abstract*—Decentralized Finance (DeFi) is an emerging paradigm within the blockchain space that aims to revolutionize conventional financial systems by applying blockchain technology. The substantial value of digital assets managed by DeFi protocols makes it a lucrative target for attacks. Despite the human resources and the application of automated tools, frequent attacks still cause significant fund losses to DeFi participants. Existing tools primarily rely on oracles similar to those used in traditional software analysis, making it challenging for them to detect functional bugs specific to the DeFi domain. Since blockchain functions as a distributed ledger system, the foundation of any DeFi protocol is the accurate maintenance of key state variables representing user funds. If these variables are not properly updated or designed to reflect the intended flow of funds, attackers can exploit these flaws to steal assets. From the study of popular DeFi protocols, we observe that, in DeFi systems, to ensure a transaction does not misappropriate someone's fund, the direction of changes (increase or decrease) of values associated with the amount of asset or debt of a user has to adhere to some fairness properties. We propose a concept called fairness bug which allows attackers to gain profit without cost. We propose an inter-procedural and inter-contract static analysis technique that utilizes symbolic execution and an SMT solver to automatically detect fairness bugs in DeFi smart contracts. We have implemented our fairness-checking approach in our tool, named FairChecker. We evaluate our tool on a benchmark of 113 real-world DeFi protocols with 34 fairness bugs. The results show that our tool can detect 32 bugs with a recall of 94.1% and a precision of 46.4%, demonstrating its effectiveness.**

***Index Terms*—Blockchain, static analysis, bug finding**

I. Introduction

Starting with the introduction of Bitcoin [1] by Nakamoto in 2008, blockchain technology has experienced rapid development and adoption in recent years. As of Aug. 2024, the market capitalization of blockchain has reached 2.2 trillion US Dollars [2]. Smart contracts are programmable agreements that execute automatically on a blockchain without the need for external supervision. Bitcoin's scripting language allowed for some limited forms of smart contracts, such as time-locked transactions. Ethereum and other modern blockchain platforms such as Solana extend beyond simple transactions, allowing developers to build complex applications on its blockchain, utilizing smart contracts for various purposes. Among all the applications of smart contracts, decentralized finance (DeFi) is one of the most popular and sought-after services. The inherent distributed and tamper-resistant nature of smart contracts has rendered it a critical infrastructure for the DeFi ecosystem. DeFi offers open access, transparency, and financial inclusion, allowing users to access various financial services and applications autonomously. DeFi protocols have garnered substantial user adoption, reaching a peak of $181 billion in total digital assets under management [3].

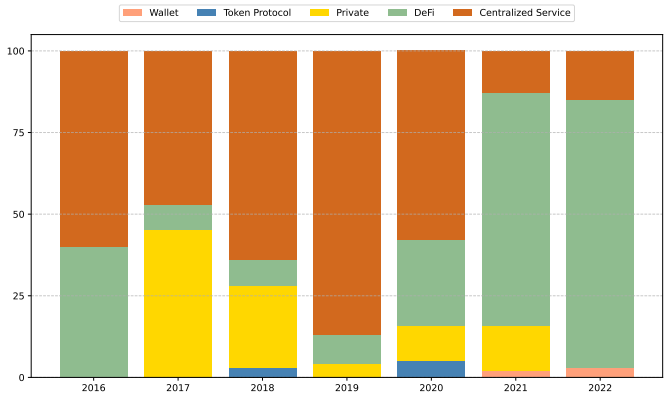


Fig. 1: Trend of crypto stolen in hacks by victim platform types (2016-2022).
Data source: Chainalysis[4]

However, the substantial value of digital assets handled by DeFi protocols renders them an enticing target for attacks. Since 2021, DeFi has become the primary target of crypto hackers, with the trend intensifying after. Figure 1 shows the trend of the most affected types of platforms by hacks. DeFi protocols as victims accounted for 82.1% of all cryptocurrency stolen by hackers in 2022 — a total of $3.1 billion — up from 73.3% in 2021. Due to the nature of smart contracts and DeFi, software vulnerabilities and financial loss are inherently coupled. To make the situation worse, smart contracts are immutable once deployed, making it hard to patch or upgrade them. One of the most notorious attacks on DeFi is the DAO attack [5] on Ethereum, which caused over $60 million fund loss for the participants. To protect their products, the DeFi community has invested a huge amount of resources in either developing automated tools or hiring ethical hackers to audit the code, aiming to find bugs before the launch of the project. In the first half of 2022, over $14 million has been paid to ethical hackers for auditing the code. Many automated

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00134

tools, such as Sailfish [6], Manticore [7], and Slither [8] have also been developed by academic researchers or commercial companies. However, it's hard to say that the current efforts and tools are enough. During the first half of 2022, the DeFi ecosystem suffered a staggering loss of over $265 million [9].

In smart contracts, state variables are used to store persistent data that is critical to the operation and state of the contract. These variables are stored on the blockchain and remain available throughout the lifetime of the contract. This persistence ensures that the contract can maintain its state between function calls and transactions. DeFi applications, in particular, depend on state variables to record the financial status of users. For example, an ERC-20 contract usually uses a state variable `_balances`, which is a mapping from address to unit, to record the number of tokens each user has. Maintaining the integrity of those state variables is critical to prevent misuse of user funds. However, automatically verifying the integrity of state variables in DeFi smart contracts presents significant challenges due to the diverse and complex implementations across different contracts. Each DeFi application may have unique logic for updating state variables, incorporating a variety of financial models, business rules, and interaction patterns. This variability complicates the task of creating a general verification method.

We observe that DeFi protocols, similar to traditional financial systems, highly rely on the integrity of the bookkeeping system. Consider some situations in traditional financial activities. When you deposit cash at a bank, you expect the balance in your account to increase by the value of your cash. Otherwise, the bank has possession of your money, and you have no way to prove your right to get the money back. Similarly, when you borrow money from a bank, e.g., using a credit card, the bank also keeps track of the balance of the money you owe. Any rational legitimate parties, including individuals or other contracts participating in the protocol, would require the value representing their share or liability in the protocol to move in the correct direction after the execution of a transaction.

Building on our earlier observations, we propose two fundamental fairness principles applicable to all legitimate DeFi protocols. First, no participant should be able to increase their share within a protocol without incurring a cost. Second, it should be impossible for anyone to reduce another participant's share without compensating them. We propose a method to infer critical variables that represent balances automatically. Then, we design a symbolic execution system to explore the execution paths of every external function and collect information on modifications made to those critical variables. The constraints, the modification information, and the fairness properties are then transformed into SMT formulas and checked by an SMT solver. If a counter-example can be found that violates the fairness properties, we consider the function unfair and possibly contains a bug.

We have implemented the fairness property checker on top of the Slither static analysis framework [8], namely FairChecker. We evaluate the tool on a benchmark of 113 real-world DeFi protocols with 34 fairness bugs. The results show that our tool can detect 32 fairness bugs with a recall of 94.1% and a precision of 46.4%, which demonstrates the effectiveness of our tool. To sum up, we make the following contributions:

- We present a study on popular DeFi protocols and define two general fairness properties on DeFi protocols.
- We propose an approach to detect fairness bugs in smart contracts, combining an annotation mechanism, a symbolic execution engine, and an SMT solver-based validator that verifies the two defined fairness properties.
- We implement the proposed approach as a tool, namely FairChecker to automatically detect fairness bugs in smart contracts.
- We evaluate our approach on real-world DeFi protocols, showing its effectiveness and efficiency in detecting fairness bugs.

Paper organizations. The rest of the paper is organized as follows. Section II explains the necessary background about the basic components of DeFi Protocols. Section III introduces fairness bugs and our detection approach by examples. Section IV formally defines fairness properties. Section V presents the design and implementation details of our technique. Section VI presents our evaluation. Section VII talks about the limitations of our technique. We discuss related works in Section VIII and conclude the paper in Section IX.

II. Background

Address. In Ethereum, an address is a 20-byte (160-bit) hexadecimal value that serves as a unique identifier for an account on the Ethereum blockchain. An address can be associated with a user-controlled account. It can also be a contract account address, which is created when a smart contract is deployed on the Ethereum blockchain.

Tokens. Ether (ETH) is the inherent cryptocurrency of Ethereum blockchain. It serves the purpose of rewarding miners and validators for adding blocks to the blockchain, as well as functioning as a payment method, which includes covering transaction fees. To enable more complex business models, smart-contract-enabled tokens, like ERC tokens, are introduced in Ethereum. ERC stands for "Ethereum Request for Comments" [10]. ERC standards define a common set of functions and interfaces that ensure compatibility and interoperability among different tokens. For example, ERC20 defines common behaviors of fungible tokens, which are interchangeable. ERC721 defines unique non-fungible tokens. ERC tokens can be created, transferred, or destroyed from a central contract. Tokens in the DeFi space can be considered analogous to currency or stocks in traditional finance.

III. Motivation

In this section, we first introduce our observations of fairness in two of the most popular DeFi applications. Then, we illustrate how violations of fairness can cause critical bugs by real-world examples followed by a sketch of our approach to detect them.

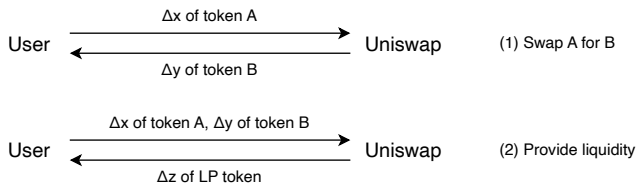


Fig. 2: Token flows in Uniswap

A. Fairness in Popular DeFi Protocols

Most financial activities in DeFi involve token exchanges, which can be classified into two main categories: (1) Direct token exchanges, where users swap one or more types of tokens for different tokens within the same transaction (e.g., trading token A for token B); and (2) Indirect token exchanges, where users transfer tokens to a contract without immediate returns (e.g., depositing tokens to earn interest and withdrawing later). This subsection discusses these norms with examples and presents our findings on fairness.

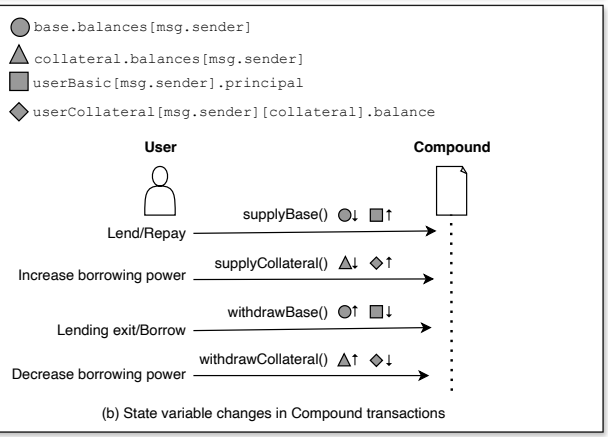
Direct exchange of tokens. Take Uniswap as an example. Uniswap is one of the most popular DeFi trading applications. It is a decentralized exchange (DEX) protocol that allows users to trade ERC-20 tokens without the need for a central authority. It provides the following major services for users:

- **Token swaps.** Uniswap enables users to exchange one ERC-20 token for another directly. It uses liquidity pools, which hold both tokens, instead of traditional order books to execute trades, ensuring continuous liquidity.
- **Providing Liquidity.** Users can become liquidity providers (LPs) by depositing an equal value of two tokens into a liquidity pool. In return, LPs receive liquidity tokens that represent their share of the pool, which they can redeem at any time.

Figure 2 shows the flow of tokens in the two core functions provided by Uniswap. For (1), the user transfers Δx of token A to the Uniswap contract. The contract calculates the number of token B to be sent to the caller using an automated market maker (AMM) model based on the constant product formula $x \times y = k$, where k is a non-zero constant, x is the number of token A in the contract, and y is the number of token B in the contract. With the newly transferred Δx of token A, the number of returned tokens B is $\Delta y = y - \frac{k}{x+\Delta x}$. From the formula for Δy, we can see that whenever Δx is non-zero, Δy is guaranteed to be non-zero. This means that for any transaction of the swap function, when the state variable representing the number of token A the user has (`A._balances[user]`) decreases, another state variable (`B._balances[user]`) must increase. For (2), the returned number of LP tokens is calculated by the formula $\Delta z = z \times \min\left(\frac{\Delta x}{x}, \frac{\Delta y}{y}\right)$, where z is the supply of LP tokens before the new liquidity is provided. Similar to (1), when users transfer a positive number of token A and token B to the contract, the returned number of LP tokens must be non-zero. We can see that for the caller of these

```
1  function supplyCollateral(address asset, uint128 amount) {
2    ERC20(asset).transferFrom(msg.sender, address(this), amount);
3    uint128 collateral = userCollateral[msg.sender][asset].balance;
4    uint128 collateralNew = dstCollateral + amount;
5    userCollateral[msg.sender][asset].balance = collateralNew;
6  }
7  function withdrawBase(uint256 amount) {
8    // Accure interest
9    accrueInternal();
10   UserBasic memory srcUser = userBasic[msg.sender];
11   int104 srcPrincipal = srcUser.principal;
12   int104 srcPrincipalNew = srcPrincipal - amount
13   updateBasePrincipal(msg.sender, srcUser, srcPrincipalNew);
14   // Check if the user has enough collateral
15   if (!isBorrowCollateralized(msg.sender))
16      revert NotCollateralized();
17   ERC20(asset).transfer(msg.sender, amount);
18 }
```

(a) Compound core functions



(b) State variable changes in Compound transactions

Fig. 3: Balance variable changes in Compound functions

functions, the decrease and increase of their balances must happen simultaneously. We call such functions fair functions.

Indirect exchange of tokens. Compound is an algorithmic, autonomous interest rate protocol built on the Ethereum blockchain. It provides lending services for users. Each pool contract in Compound manages a pair of tokens, the base token, and the collateral token. It allows users to deposit the base token and earn interest, or to borrow the base token against their deposited collateral. Figure 3(a) contains a code snippet of two core functions of Compound. Different from Uniswap, the exchange of base and collateral tokens doesn't happen in the same transaction. The contract maintains two state variables, userBasic and userCollateral, which store the number of borrowed base tokens and deposited collateral tokens respectively. Figure 3(b) shows the changes in balance variables when users interact with Compound contracts. The function `supplyCollateral` allows users to deposit collateral tokens to the contract. The number of collateral tokens a user deposited in the contract will be used to calculate the borrowing power of users. The collateral tokens are transferred from the caller at line 2 (▲ decreases), and the value representing the deposited tokens is updated at line 5 (♦ increases). The function `withdrawBase` allows users to borrow base tokens against their collaterals. Depending on the number of tokens the caller wants to borrow, the function updates the debt the

```
1  contract VaultTracker {
2  struct Vault {
3    uint256 notional;
4    uint256 redeemable;
5    uint256 exchangeRate;}
6  mapping(address => Vault) public vaults;
7  function exit(
8      Hash.Order calldata o,
9      uint256 a) internal {
10   Erc20 uToken = Erc20(o.underlying);
11   uint256 premiumFilled = (a * o.premium) / o.principal;
12   uToken.transferFrom(o.buyer, msg.sender, premiumFilled);
13   require(transferN(msg.sender, o.buyer, a))}
14 function transferN(address f, address t, uint256 a)
15     internal returns (bool) {
16   Vault memory from = vaults[f];
17   Vault memory to = vaults[t];
18   require(from.notional >= a);
19   // Updates of other fields. Code omitted
20   from.notional -= a;
21   vaults[f] = from;
22   to.notional += a;
23   vaults[t] = to;
24   return true;
25 }}
```

Fig. 4: Unfair example (increase of caller's fund)

caller owes at line 13 (reducing ■ by `amount`) and checks if the collateral is enough at line 15. The borrowed tokens are transferred to the caller at line 17 (increasing ● by `amount`). Functions `supplyBase` and `withdrawCollateral` perform the opposite operations of the aforementioned functions. The changes in balance variables are therefore inverse. We can see they all follow the one-up-one-down pattern just as functions in Uniswap.

In addition to Uniswap and Compound, we examined decentralized exchanges (DEXs) such as SushiSwap and Balancer, lending protocols like Aave and MakerDAO, and derivative trading platforms like Synthetix. Our analysis of the external functions accessible to users revealed that balance variable changes for the caller consistently adhere to the principle that an increase in one balance variable is always accompanied by a decrease in another balance variable. This observation leads us to conclude that the principle of fairness is a fundamental characteristic of legitimate DeFi applications.

B. Fairness bugs in DeFi

Unfair Example 1. (Increase of the Caller's Fund) Figure 4 presents a real-world example of a buggy smart contract, `VaultTracker`, from the Swivel project [11]. The code is modified for simplicity. This contract features a function `exit`, enabling callers to sell their shares in exchange for the underlying ERC20 token. The user's share is represented by the field `notional` in struct Vault. When `exit` is called, it initially transfers an amount of the `uToken` from the buyer to the caller, determined by parameter a (the share to be sold), as seen in the code at line 13. Subsequently, the function `transferN` is invoked at line 14 to adjust the notional values for the caller and the buyer. The function first creates local copies of `vaults[f]` and `vaults[t]` (lines 16-17) and

```
1  contract Vault{
2  address public override token;
3  mapping(address => uint256) public attributions;
4  function addValue(
5      uint256 _amount, address from, address beneficiary
6      ) external override onlyMarket returns (uint256 _attributions){
7    _attributions = (_amount * totalAttributions) / valueAll();
8    IERC20(token).safeTransferFrom(from, address(this), _amount);
9    balance += _amount;
10   totalAttributions += _attributions;
11   attributions[beneficiary] += _attributions;
12 }
13 function withdrawValue(uint256 _amount, address _to)
14     external override returns (uint256 _attributions) {
15   _attributions = (_amount * totalAttributions) / valueAll();
16   attributions[msg.sender] -= _attributions;
17   IERC20(token).safeTransfer(_to, _amount);
18 }}
```

Fig. 5: Unfair example (decrease of another's fund)

updates their notional values (lines 20 and 22). Finally, these local variables are written back to the state variable `vaults`, overwriting the original values. The balance changes in an `exit` transaction are as follows:

$$(1) \uparrow uToken.balances[msg.sender]$$
$$(2) \downarrow uToken.balances[o.buyer]$$
$$(3) \downarrow vaults[msg.sender].notional$$
$$(4) \uparrow vaults[o.buyer].notional$$

At first glance, this seems fair, as each participant has one balance going up and another going down. However, a bug in `transferN` allows for unfair transactions. The issue arises when `f` and `t` are the same; the update at line 21 is overwritten by line 23. In a case where `msg.sender` and `o.buyer` are identical, balance changes (1) and (2) cancel each other out, as transferring to oneself effects no change. Moreover, (3) gets overwritten by (4), resulting in the sole balance change being an increase in the caller's notional value. This example demonstrates how an unfair increase in the caller's balance can indicate a bug.

Unfair Example 2. (Decrease of Another's Fund) Figure 5 presents a buggy code from `Vault` in the Insure project [12], slightly modified to illustrate the core concept more clearly. This contract uses the `attributions` mapping variable to track user funds. It allows users to deposit and withdraw funds. Unlike the fair example's deposit function, which only transfers funds from the caller, the `addValue` function enables a caller to transfer funds on behalf of the user `from` and increase the `attributions` balance of `beneficiary`. This function is restricted to certain privileged addresses via the `onlyMarket` modifier. Consider a scenario where a privileged caller `marketer` invokes `addValue`. It's assumed that user `from` has authorized `marketer` to spend a specified `amount` of `ERC20(token)`. The function executes `transferFrom` to move these tokens from `from` and boosts the `attributions` balance of `beneficiary`. This leads

to two balance changes:

$$(1) \uparrow Vault.attributions[beneficiary]$$
$$(2) \downarrow ERC20(token).balances[from]$$

This transaction is deemed unfair if $from \neq beneficiary$ and $from \neq msg.sender$, as the `from` address's balance in `ERC20(token)` decreases without any compensatory increase, resulting in a net loss for address `from`. An attacker could exploit this by setting `beneficiary` to their own or a controlled address, and after executing `addValue`, they could use `withdrawValue` to extract the misappropriated funds at no cost. Even though the function is protected by the modifier, a compromised privileged user can put all user's funds at risk.

Checking fairness properties. From the two unfair examples above, we have two observations about the fairness of transactions. Firstly, a transaction that allows the caller to increase its balance without any cost might indicate a bug. Secondly, a transaction that reduces someone's balance without compensation could also signify a bug. We can derive two fairness properties, respectively. To perform the automated validation on the two properties, we need to collect the balance changes as demonstrated in the two examples. We propose a method based on symbolic execution. To detect the bug in function `addValue`, we explore the CFG of the function `addValue`. For each path, we collect the symbolic changes to the balances. One path generates the balance changes as we presented in the discussion of **unfair example 2**. We then feed the path constraints, balance changes, and encoded fairness properties to an SMT solver. It can find an assignment that violates the properties, for example, `from = 0x1`, `beneficiary=0x2`, `msg.sender=0x2`, and `_amount=10`. The bug is successfully detected.

IV. Fairness definitions

In this section, we formally define balance annotations, fairness properties, and other concepts necessary to define fairness bugs.

Definition 1.(Balance). A balance is a numerical value stored in a state variable representing the number of tokens that can be withdrawn by a specific address (asset of the address), the number of tokens owed by a certain address to this contract (debt of the address), or other numerical values that represents user's share in the contract. It serves as a record of the available token balance for individual addresses participating in the contract. Balance can be denoted by a tuple of three attributions.

$$balance = \langle identifier, type, owner \rangle$$

The meaning of each attribution is as follows:

- Balance identifier: A reference to a balance value stored in a state variable.
- Balance type: The type of the balance. Its value can be 'Asset' or 'Debt'.
- Balance owner: A reference to the address associated with the balance.

A balance identifier can be as simple as just the symbol of the state variable if the variable is a `uint` storing the balance value. For example, the cod in Figure 6 implements a wallet that only allows a fixed address to deposit funds. The balance in this contract can be represented as $\langle balance, Asset, owner \rangle$, meaning that variable $balance$ represents the amount of asset owned by the address stored in variable $owner$.

```
uint256 public balance;
address owner;
function deposit(uint256 amount) external {
  require(msg.sender == owner);
  IERC20(token)
    .transferFrom(msg.sender, address(this), amount);
  balance += amount;
}
```

Fig. 6: A simple wallet

When the balance value is stored in a data structure like mapping or struct, the balance identifier can be represented using a template of references that refer to the mapped value or field. For example, the balance in the example of Figure 4 can be denoted as $\beta = \langle contributions[OWNER], Asset, OWNER \rangle$, meaning that the values stored in the mapping variable `contributions` are asset balance values, the balance owner is the key used to get the value. Note that, `OWNER` in `contributions[OWNER]` is not an identifier of any variables in the contract. It is a placeholder that represents the key used to get the mapped value. For the example depicted in Figure 5, the 'notional' field within the 'Vault' struct represents the amounts of assets that a user owns in the contract. The balance variable ownership entry can be denoted as follows:

$$\beta = \langle vaults[OWNER].notional, Asset, OWNER \rangle$$

Definition 2. (Balance Change). A balance change is the modification of a balance within a smart contract during transaction execution, representing the difference between the new balance value and the old balance value. We denote it as $\delta = \langle \beta, \texttt{diff} \rangle$, where β is a balance as defined in **Definition 1**, `diff` is the symbolic value representing the difference.

Given the definition, the balance changes in the `addValue` function in the fair example described in Figure 4 can be represented as follows:

$$\delta_1 = \langle attributions[beneficiary], Asset, from, \_attributions \rangle$$
$$\delta_2 = \langle ERC20(token).balances[from], Asset, from, -amount \rangle$$

Fairness property 1. In a transaction, a balance owned by the caller of a transaction can increase if and only if another balance owned by the caller decreases.

We denote a transaction as $t = \langle caller, \Delta \rangle$. $caller$ is the caller of the transaction (i.e., msg.sender). $\Delta = \{\delta_1, ...\}$ represents the set of all balance changes. In the context of our definitions and the analysis we are going to perform, an

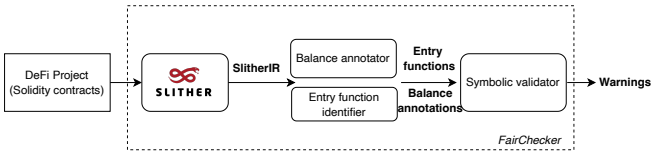


Fig. 7: Workflow of FairChecker

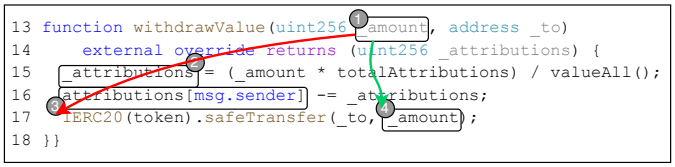


Fig. 8: Finding balance variables

increase in a balance of type Asset is equivalent to a decrease in a balance of type Debt, and vice versa. To simplify the expression, we only define the case when the caller increases its asset balance. The formal definition of this rule is as follows:

$$\forall \delta = \langle i, o, t, d \rangle \in \Delta, o = caller \wedge t = Asset \wedge d > 0$$
$$\Rightarrow \exists \delta' = \langle i', o', t', d' \rangle,$$
$$o' = caller \wedge ((t' = Asset \wedge d' < 0) \vee (t' = Debt \wedge d' > 0))$$

Fairness property 2. In a transaction, a balance owned by an address other than the caller can decrease if and only if another balance owned by that address increases.

Given a transaction $t = \langle caller, \Delta \rangle$. This rule can be formally defined as:

$$\forall \delta = \langle i, o, t, d \rangle \in \Delta, o \neq caller \wedge t = Asset \wedge d < 0$$
$$\Rightarrow \exists \delta' = \langle i', o', t', d' \rangle,$$
$$o' = o \wedge ((t' = Asset \wedge d' > 0) \vee (t' = Debt \wedge d' < 0))$$

Please notice that, while it is unfair for the balance of a non-caller address to decrease without having a balance of it to increase, we consider it legal for a caller's balance to decrease solely. Because in this case, the caller is not profitable and no other's fund is at risk. Therefore an attacker would never perform such a transaction. It is generally possible for a caller to deliberately decrease their balance without getting paid back in other forms. For example, one can transfer tokens to another address that is controlled by themselves. Or one can destroy their tokens by sending them to address(0).

Definition 3.(Fair transaction, fair function). A fair transaction is a transaction that fulfills both fairness properties 1 and 2. A function in a smart contract is fair if all possible transactions of the function are fair transactions.

V. Design of FairChecker

Figure 7 shows an overview of FairChecker. FairChecker takes a DeFi project (e.g., a hardhat [13] or a truffle [14] project) as input. The input usually contains multiple smart contracts. FairChecker first utilizes Slither[8] to generate Slither IRs, an intermediate representation of contracts. Then, it identifies functions in the project that are meant to be used externally. Meanwhile, FairChecker performs a taint-analysis-based method to extract balance annotation from the IRs. Using the balance annotations, FairChecker symbolically executes the functions, collecting balance changes, and validates the fairness properties defined in section IV using SMT solver [15]. At last, FairChecker reports unfair functions. Our implementation currently supports Solidity, and hence by default, supports most EVM-compatible blockchains such as Arbitrum and Polygon. For others like Solana, the concept of fairness bugs still applies, and DeFi's business logic remains unchanged. Our solution can be applied to them after adapting to different programming languages.

A. Identify entry functions

Firstly, we need to filter out all testing or mock contracts in the projects as they will not be deployed on Blockchain. We remove any contracts that are inside directories with similar names to "testing", "mock", etc. In addition, we filter out any contracts imported from external projects, such as OpenZeppelin [16]. For the contracts under analysis, we find that not all functions defined as public or 'external' can be called by users. For example, contract A may have an external function `foo` which can only be called by contract B in its function `bar`. It can be achieved by restricting the access to `A.foo()` with a modifier that checks if `msg.sender == admin`, where admin is set to the address of contract B. This is a common design pattern used by DeFi projects to separate roles among contracts. It is important to filter out `foo` because if balance changes occur in both `foo` and `bar`, analyzing `foo` alone may cause false positives. To filter out such functions, we first find restricted external functions by doing a pattern matching on statements like `require(msg.sender==v)` or `require(v[msg.sender])`. Then, we try to find cross-contract callers of such functions by matching the function signatures. If any caller can be found, we conclude that the restricted external function cannot be called by users and filter it out. After performing those filters, we can feed the remaining functions to the symbolic validator.

B. Balance annotator

To facilitate the automated analysis of the fairness of smart contract functions, we need to first create balance annotations as described in section IV. Although it is possible to manually annotate the contracts, the task requires a lot of manual effort and it's prone to mistakes. We propose a heuristic-driven method to automatically generate balance annotations.

ERC token contract. For contracts that implement ERC interfaces such as ERC20, conventionally state variables like '_balances' are used for storing balance values. For example, we can automatically extract the balance defined in ERC contracts as follows:

- ERC20: ⟨`_balances[OWNER], Asset, OWNER`⟩
- ERC721: ⟨`_balances[OWNER][ID], Asset, OWNER`⟩

While names like '_balances' are generally used by the balance variable in ERC tokens, it is not forced by the standard or interfaces. In such cases, we analyze the function

TABLE I: Balance change generation rules

Rule	Syntax	Balance changes
Low-level call	`target.call{value: x}(data)`	⟨this, Ether.balances[this], -x⟩ ⟨target, Ether.balances[target], +x⟩
High-level call	`target.func{value: x}(args)`	⟨this, Ether.balances[this], -x⟩ ⟨target, Ether.balances[target], +x⟩
Fungible ERC token transfer	`token.transfer(to, x)`	⟨this, ERC20[target].balances[msg.sender], -x⟩ ⟨to, ERC20[token].balances[to], +x⟩
Fungible ERC token transferFrom	`token.transferFrom(from, to, x)`	⟨from, ERC20[token].balances[from], -x⟩ ⟨to, ERC20[token].balances[to], +x⟩
Non-fungible ERC token	`token.transferFrom(from_addr, to_addr, id)`	⟨from, ERC721[token].balances[from], -1⟩ ⟨to, ERC721[token].balances[to], +1⟩

`_transfer`, which has to be implemented by any ERC token contract to find balance variables used by statements like `var[msg.sender] -= amount`. It is achieved by pattern matching of code.

Non-ERC token contract. For non-standard contracts, we utilize the following heuristic for identifying balance variables. If a numerical value in a parameter can flow to both a token transfer instruction (e.g., `ERC20.transfer()`) and a state variable store operation, the state variable is likely a balance variable. If the state variable is a mapping variable, the index that uses `msg.sender` is likely the owner of the balance.

The heuristic is implemented in an inter-procedural taint analysis. The source is any numerical parameters (e.g., uint or int). There are two types of sinks. One is ERC token transfer function calls and the other is storage operation of state variables. For a source, if it flows to both types of sinks, we consider the state variable modified at the sink point candidates of balance variables. Figure 8 demonstrates the approach with the example code from Figure 5. The function `withdrawValue` allows users to withdraw a certain amount of tokens from the contract and decrease the caller's share in the contract accordingly. The function has one numerical parameter `_amount`. We start propagating it along the program flows. `_attributions` is tainted at line 15. Then it taints `attributions[msg.sender]` through a store operation of the state variable. Finally, `_amount` flows to the `ERC20.transfer()` function call at line 17. Therefore, it flows to both types of sinks. As `msg.sender` is used as the index of the state variable reference, we consider `attributions[OWNER]` as a balance reference, and the owner of the balance is the index used in the mapping. Lastly, we need to determine if the balance is asset or debt. We first utilize a keyword-based approach to match the state variable names or field names with commonly used asset/debt names, such as 'balance', 'share', or 'debt'. For names that don't contain any of those keywords, we utilize an NLP sentiment analysis package NLTK [17] to check if it's a negative or positive word and assign balance variables with negative names as debt.

C. Symbolic Execution

FairChecker implements a symbolic execution engine based on Z3. Fix-sized elements such as uint and address are modeled using Z3's fixed-size bit-vector expression. Variable-length elements such as array and mapping are modeled using Z3's array expression. Compound elements such as structs are modeled using Z3's DataType expression. Given an entry function, FairChecker executes it symbolically and collects path constraints and balance changes. The balance changes are then used to encode the two fairness properties. Together with path constraints, they are sent to the Z3 solver to check satisfiability. If a counter-example can be found, meaning that the transaction is unfair and we find a fairness bug, we halt the analysis and output the model assignment generated by the solver. Otherwise, we repeat the above process until path exploration is done or the time budget is used up.

Collect implicit balance changes For balance variables being annotated, the balance changes can be represented by subtracting the balance variable from the updated balance variable after the symbolic execution (`new_balance - old_balance`). However, three types of instructions need extra attention in symbolic execution as they can lead to implicit balance changes, including low-level or high-level calls with value, ERC token function calls, and external DeFi protocol calls. The rules for generating balance changes from the first two types of instructions are shown in Table I. Note that, all the balance changes generated by those instructions are changes on balance of type Asset, thus in the associated rules, the Asset notion is omitted.

Low-level call or high-level call with native token transfer. The low-level call in Solidity uses `call` function to execute transactions. It calls the function encoded in the parameter `data` and transfers Ether to the target address. For example, the instruction `target.call{value: x}()` transfers `x Wei` from the caller address to the address `target`. Ether, as a native token, is not implemented by smart contracts. Therefore there are no balance variables at the smart contract level that hold the balance values. We use a global array variable `Ether.balances` to represent the balances of native tokens for each address. Similar to a low-level call, a high-level call to an external function in other contracts can take extra argument `value` in order to transfer Ether to the target while executing the external call. The balance change here is handled the same way as low-level calls

ERC token function calls. In DeFi protocols, ERC token transfer is one of the most common operations that lead to balance changes. In our symbolic execution, for ERC contracts that are not implemented in the project, we don't go into the implementations when analyzing `transfer` function calls.

We use a global symbolic variable ERC.balances to track the changes. There are two types of transfer functions:

- `transfer(to, amount)`. This function allows transferring tokens from the caller address of the function to an address passed to the function as an argument. `to` is the address of the recipient of the tokens. `amount` is the number of tokens being transferred. The corresponding balance changes are (1) The ERC token balance owned by the contract itself that calls `transfer` decreases by `amount` and (2) The ERC token balance owned by the recipient increases by `amount`.
- `transferFrom(from, to, amount`. It takes an additional argument `address from` that represents the address from which the tokens are being transferred. Therefore the balance decrease can happen to an address other than the caller. Our handling of `transferFrom` is similar to `transfer` except for this difference.

For non-fungible ERC tokens such as ERC721, `transferFrom` also takes an additional argument `tokenId` to represent the unique id of the NFT being transferred. In our analysis, we don't differentiate NFT tokens. We are only interested in the total number of tokens one owns. Therefore, we consider that an NFT transfer results in the corresponding balance changing by +1/-1.

External DeFi protocol calls. Except for external ERC token contracts, many DeFi protocols depends on other external services such as Uniswap. Although in some cases, it is possible to obtain the implementation of the external contracts and perform analyses as if they are part of the project, it is generally not necessary as they are not targets for bug detection. As we are only interested in external calls that cause balance changes, we model the ERC token transfers or swaps when an external call such as Uniswap's `swapExactTokensForTokens` which exchanges two tokens for the caller. The rules to generate balance changes are similar to how we handle ERC transfer functions.

Resolve calls to interfaces As our goal is to perform an inter-procedural and inter-contract analysis, we need to obtain the caller-callee information at each call site. For calls to another function within the same contract, the callee is fairly easy to determine. The challenging part is to find the callee of a call to an external function in a different contract. If the target contract is known, the callee can also be easily determined by looking up the implementation of the target contract. The more complex case is when the target contract implements interfaces or abstract contracts and we only know the abstract type of the target contract. In this case, we have to resolve the actual place of implementation of the interface. We first try to find an implementation within the project. If an implementation can be found, we consider it the target callee. If more than one implementation exists, we consider all of them to be possible targets of the call. During symbolic execution, we fork the state at such a point to explore all possible targets.

Accelerate SMT solving. When developing the tool. We find that many contracts involve complex computations, which makes it hard for the SMT solver to solve the constraints in a reasonable time. We alleviate the issue with some heuristics. First, many math operations are on 256-bit integers in smart contracts, which makes solver hard to solve. We use 32-bit vectors to represent integers wider than 32 bits to reduce the solving time. Starting with Solidity 0.8, integer overflow/underflow are automatically checked at runtime, making these bugs rare in recent contracts. Therefore, addressing such low-level issues is not a focus. When constructing SMT constraints, we ensure that math operations are safe (no overflow/underflow). With the absence of overflow/underflow at runtime, 32-bit vectors are sufficient in model math operations in Solidity. Also, some older contracts use the SafeMath library from Openzepplin to avoid overflows. Going into the implementation of those libraries greatly increases the size of the constraints as they usually contain many branches. We can safely replace them with operations provided by SMT solver. Starting with Solidity 0.8, integer overflow/underflow are automatically checked at runtime, making these bugs rare in recent contracts. When constructing SMT constraints, we ensure that math operations are safe (no overflow/underflow). With the absence of overflow/underflow at runtime, 32-bit vectors are sufficient in modeling math operations in Solidity.

TABLE II: Occurance rate of fairness bugs

benchmark	#Proj	#Unfair Proj	#Bug	#Unfair bug
Web3bugs	113	24 (21.2%)	462	34 (7.4%)
DefiLlamaBugs	151	10 (6.6%)	155	10 (6.4%)

VI. Evaluation

To explore the capability of our proposed approach in this paper, we evaluated FairChecker to answer the research questions below.

- RQ1: How common are fairness bugs in the real world?
- RQ2: How effective does FairChecker detect fairness bugs of smart contracts?
- RQ3: What are the root causes of fairness bugs, and can they be detected by existing tools?

Benchmark. For the benchmark, our objective is to utilize real-world bugs with ground truth. To achieve this, we employed the Web3Bugs benchmark curated by Zhuo et al. [9] from the esteemed audit contest platform, Code4Rena [18]. The dataset comprises 113 smart contract projects and 462 high-risk bugs, all confirmed by the respective developers. In addition, DefiLlama [3] maintains a page collecting real-world hacks on DeFi protocols starting from 2016. We sampled the 155 bugs from the last two years (from May 2022 to May 2024) as part of our benchmark.

Environment. All the experiments are run on a Ubuntu-20.04 server equipped with a 12-core 20-thread Intel Core i5 CPU, with 3.60GHz speed and 64GB of RAM.

TABLE III: Effectiveness of FairChecker on the benchmark

ID	Name	LoC	Ext. funcs	UNFAIR	TP	FP	Precision	Recall	Time(s)
5	Vader Protocol	2123	184	2	2	0	100%	100%	1169
13	Reality Cards	3043	195	1	1	0	100%	100%	2488
16	Tracer	2453	88	1	1	1	50%	100%	703
23	Notional	12654	68	1	1	1	50%	100%	657
39	Swivel	916	20	2	2	2	50%	100%	160
42	Mochi	1562	93	2	2	1	66.7%	100%	1069
44	Tally	263	32	1	1	0	100%	100%	320
52	Vader Protocol p2	5018	45	4	3	1	75%	75%	482
61	Sublime	4457	243	1	1	2	33.3%	100%	1687
64	PoolTogether	1272	50	1	1	1	50%	100%	368
69	NFTX	6503	282	1	1	1	50%	100%	2512
70	Vader Protocol p3	6346	43	2	2	0	100%	100%	460
71	InsureDAO	2933	146	1	1	1	50%	100%	1997
78	Behodler	4793	229	2	2	0	100%	100%	2524
81	Notional	1115	18	1	1	0	100%	100%	273
83	Concur Finance	985	34	1	1	2	33.3%	100%	194
90	Phuture Finance	1471	40	1	1	0	100%	100%	227
96	Timeswap	7364	46	1	1	0	100%	100%	232
97	Biconomy Hyphen 2.0	2420	116	1	1	2	33.3%	100%	1250
103	LI.FI	1596	29	1	1	0	100%	100%	551
104	Joyn	778	33	2	2	1	66.7%	100%	802
115	Mimo DeFi	7121	383	1	1	0	100%	100%	3210
122	Cally	438	137	2	2	0	100%	100%	1525
125	Sturdy	542	24	1	0	0	-	0%	296
Projects without fairness bugs				0	0	21	-	-	
Overall				34	32	37	46.4%	94.1%	

A. RQ1. Prevalence

To find out how common the fairness bugs are, we manually examine all the bug reports in benchmarks Web3Bugs and DefiLlamaBugs. For each bug report, we examine the provided exploit trace and check if (1) the exploit causes fund leakage; and (2) the attacker pays nothing in the exploit. If both criteria are satisfied, we consider it a fairness bug. Two people were involved in the manual analysis. The bug reports are evenly divided between them for checking. The results are then cross-checked by each other. The results are shown in Table II. We find that 34 bugs in Web3bugs and 10 bugs in DefiLlamaBugs are fairness bugs, which account for 7.4% and 6.4% of the total bugs, respectively. In DefiLlamaBugs, 3 bugs violate both fairness properties, and 7 bugs violate property 1 only. In Web3Bugs, 12 bugs violate both, 18 bugs violate property 1 only, and 4 bugs violate property 2 only. The 10 bugs in DefiLlamaBugs caused a total financial loss of $123.4M. These numbers demonstrate that fairness bugs are common in real-world DeFi applications and can lead to significant financial losses. The results highlight the critical need for detecting and preventing fairness bugs to safeguard user funds.

B. RQ2. Effectiveness

To evaluate the effectiveness of FairChecker. We conduct experiments on the two benchmarks. For each bug, we read the audit report and identify the entry functions that trigger the bug. Then we run FairCheckeron the projects. If a warning is reported for a buggy function, we check the counter-example generated by SMT solver and see if it does trigger the fairness bug. Table III shows the bug detection result of our tool on the Web3Bugs benchmark. In this table, the column labeled UNFAIR specifies the count of fairness bugs within the project. The columns TP and FP refer to true positives and false positives respectively. For the 24 projects that have fairness bugs, we list each one in a row in detail. For the projects without fairness bugs, we combine the results in one row. Due to the nature of symbolic execution, the exploration of paths may not stop given a reasonable time budget. We allow the analysis to run for up to 2 minutes on each function. Our tool reports 69 warnings, among which 32 are true positives and the rest are false positives. FairChecker missed only 2 fairness bugs and achieved 46.4% precision rate and 94.1% recall.

TABLE IV: Detect real-world hacks

Bug	Loss	Detected
LEVEL [19]	$1M	✔
SUPER SUSHI SAMURAI [20]	$4.6M	✔
MINER [21]	$0.4M	✔
GYMNET [22]	$2.1M	✔
METER IO [23]	$7.7M	✔
Qbridge [24]	$80M	✔
Socket [25]	$3.3M	✔
OMM [26]	$1.89M	✔
Rubic [27]	$1.41M	✔
Transit Swap [28]	$21M	✘ (no source code)

We also run FairChecker on the DefiLlamaBugs benchmark to see if it has the potential to prevent real-world hacks. To perform the experiment, We first collect the source code of the 10 buggy contracts from websites like Etherscan [29]. One of the buggy contracts is not verified on such websites, therefore

TABLE V: Root causes of fairness bugs

Label	Description	Count	DB
S2	ID-related violations	4	✘
S3	Erroneous state updates	2	✘
S4	Business-flow atomicity violations	3	✘
S5	Privilege escalation and access control issues	5	✘
S6	Erroneous accounting	2	✘
SE	Broken business models	4	✘
L6	Arbitrary external function call	2	✔
L8	Revert issues	4	✘
O1	Malicious privileged users exploitable	4	✘
O5	Typo or trivial bugs	2	✘

the source code is unavailable. Then, we run FairChecker on the buggy contract to see if it could detect the bug. Table IV shows the result. FairChecker successfully detected all 9 fairness bugs with source code available.

C. RQ3. Root causes of fairness bugs

To gain a deeper understanding of the root causes of fairness bugs and the capability of our approach, we analyzed the bug reports of the 32 detected fairness bugs in the Web3Bugs dataset. Web3Bugs dataset categorized these bugs based on their root causes. By adopting the class labels and descriptions from their classification, we have identified that the fairness bugs fall into the categories listed in Table V. Column 'DB' represents if existing tools can detect the type of bug. Labels beginning with 'S' denote bugs that require high-level semantic oracles for detection. Labels starting with 'L' represent low-level bugs, detectable using simple oracles without a deep understanding of code semantics. Bugs with labels starting with 'O' are bugs that don't fall into the previous two categories. Bugs in the 'S' categories are challenging to detect automatically. According to the study in Web3Bugs[9], no existing tools can effectively detect bugs in those categories. We run Slither [8] and Oyente [30], two state-of-the-art commercial tools, on the dataset. They are only able to detect the two 'L6' bugs. Our approach, however, successfully detected 20 bugs in 'S' categories, demonstrating the potential of using fairness properties as a general oracle for detecting high-level functional bugs.

VII. Limitations

We discuss the limitations of FairChecker, ways to mitigate them in practice, and possible future works.

A. False Positives

We analyzed the false positives (FPs) observed in the experiment described in RQ2 and identified their root causes along with possible mitigation strategies.

Root Causes. A review of the 37 FPs from the Web3Bugs benchmark revealed two primary causes. The first is inaccurate balance annotations, accounting for 32% of the FPs. This issue arises from the heuristic-based nature of the annotator, which introduces inaccuracies. The second cause is infeasible exploits, which constitute 51% of the FPs. In some protocols, the execution of a transaction may depend on preceding transactions (e.g., function B can only be executed after function A). As FairChecker currently performs per-transaction analysis, it cannot capture preconditions derived from prior calls. Consequently, FairChecker may generate infeasible value assignments for the symbolic state. The remaining 17% of FPs stem from protocol-specific issues that do not follow common patterns.

Ways to Mitigate. The inaccuracy of the annotator can be addressed in several ways. In in-house development, where developers are familiar with their projects, they can provide accurate annotations or refine auto-generated ones. In scenarios involving third-party contract audits, enhancements such as LLM-based annotations can improve accuracy, which we plan to explore in future work. For FPs caused by infeasible exploits or other reasons, preventing them entirely at the reporting stage remains challenging. However, FairChecker outputs counterexamples that include input values and state variables values triggering the bug, simplifying verification. This feature makes it straightforward to validate warnings in a properly configured mock environment, readily available during development.

B. Detecting Bugs Involving Costs to Attackers

Our approach currently identifies strictly unfair bugs involving no-cost exploits. However, many real-world exploits, such as price manipulation attacks, require attackers to spend tokens initially to gain disproportionate rewards. One major challenge is distinguishing legitimate profit-making transactions from malicious ones. Extending our approach to address this issue would require defining numerical thresholds to differentiate the two (e.g., benign transactions cannot generate more than $t\%$ profit). Such thresholds could be derived by analyzing historical transactions of similar protocols. Addressing this challenge will be the focus of our future work.

VIII. Related Work

In this section, we discuss the related work in the field of smart contract analysis and how they are related to our work.

A. Symbolic Execution and Fuzzing

One widely adopted approach for bug detection involves using bug-finders based on symbolic execution or fuzz testing. Our approach also falls into the category of symbolic execution bug detectors. Several tools, including Oyente [30], Gasper [31], Osiris [32], Manticore [7], Maian [33], teEther [34], sCompile [35], Sailfish [6], SmarTest [36], Mar [37], and Mythril [38], employ symbolic execution of EVM bytecode to identify various bug patterns, such as arithmetic bugs. Oyente, being the first symbolic execution tool for Ethereum smart contracts, is adept at detecting a range of bugs. Mythril is a well-known open-source tool that can detect various types of bugs through symbolic execution. In particular, Osiris specializes in detecting arithmetic bugs, while Maian

focuses on identifying violations of trace properties. Additionally, Gasper utilizes symbolic execution to pinpoint gas-costly programming patterns. teEther and sCompile aim at finding critical execution paths that can cause security concerns. On the other hand, Reguard [39], ContractFuzzer [40], ILF [41], sFuzz [42], Harvey [43], Vultron [44] [45], ConFuzzius [46], Smartian [47], Echidna [48] [49], and xFuzz [50] employ fuzz testing to detect common security vulnerabilities like reentrancy bugs. Some of them rely on manually-written test oracles to detect violations in response to inputs generated according to blackbox or greybox strategies. Though precise, these tools lack coverage due to the nature of the fuzzing technique, which is not an issue for static analysis tools like ours. Despite the effectiveness of symbolic execution and fuzz testing in bug detection, they may not catch all critical vulnerabilities especially functional bugs. Manticore allows users to write property methods that are to be verified during the symbolic execution. However, the properties are limited to simple checks of invariants defined by users. It doesn't support encoding more complex properties such as the fairness properties we defined. Vultron proposes a domain-specific oracle to detect bugs in DeFi protocols. The oracle is that for all parties holding a single asset, the total balance should remain unchanged. While the benefits of such a general approach are evident, it may not fully accommodate modern DeFi projects that involve multi-asset operations where the total balances of a single asset may exhibit volatility. Our proposed approach, other the other hand, defines a more general oracle that accommodates more complex DeFi business models.

B. Formal Verification

Some other tools such as ECF [51], Solc-Verify [52], VeriSol [53], VeriSmart [54], Solid [55], FairCon [56], and ScType [57] employ methods that conduct exhaustive analyses using automatic program verification techniques. ECF performs verification on execution traces to detect vulnerabilities allowing callback attacks. VerSol employs formal verification methods to formally check access-control implemented in smart contracts against the designed policy with the help of user-defined assertions. VeriSmart and Solid both are specialized in verifying arithmetic properties such as proving that arithmetic operations do not lead to over- and under-flows. Those verifiers either depend on intensive user annotations or are specifically designed for arithmetic bugs, making it hard to extend them to find functional bugs. ScType [57] is path-insensitive and largely flow-insensitive as it uses a type system. It checks type inconsistencies such as an interest rate is directly added to a balance. In contrast, our analysis is path-sensitive and leverages symbolic analysis and SMT solving to check novel fairness properties. FairCon [56] utilizes formal verification to check if an auction contract is fair to all parties. It targets a different category of bugs on a specific type of contract. It requires intensive human annotations and only works on single-contract projects. Our analysis, in contrast, automates the annotation step, supports cross-contract analysis, and applies to general DeFi protocols.

C. Static Analysis

Many other tools apply static analysis techniques such as data-flow analysis to detect vulnerabilities in smart contracts. We name a few here, such as Sereum [58], NPChecker [59], SmartCheck [60], Slither [8], Zeus [61]. All of them are designed to detect one or more types of bugs with simple and clear patterns, such as reentrancy, uninitialized variables, mishandled exceptions, arbitrary write, etc. Most aforementioned tools are based on patterns and oracles similar to what is used in static analysis on traditional software. For example, reentrancy bug detection relies on an oracle that detects transaction cycles, making them broadly applicable to all contracts. However, this generality limits their ability to address domain-specific functional bugs. Depending on properties that are specific to DeFi protocols, our tool can detect functional bugs that are out of the scope of the static analysis tools. Gptscan [62] utilizes LLM to verify properties related to bugs. Static analysis only plays a small part in the technique for filtering out false positives. And the technique cannot generate detailed reports to help verify the bug. Our technique relies on path-sensitive symbolic analysis and can generate exploits (input values, etc.) once a bug is found.

IX. Conclusion

DeFi applications have experienced rapid growth in recent years. However, there's a notable lack of in-depth study on domain-specific bugs in DeFi's business logic. In this paper, we have defined two generic fairness properties that should be held by transactions on DeFi protocols. We proposed a methodology to automatically verify the properties for Solidity programs. In an experiment on real-world DeFi protocols with fairness bugs, our tool successfully detected most of the bugs, demonstrating the potential for significant cost reduction in securing DeFi projects and minimizing fund leakage risks. Currently, our approach only identifies strictly unfair bugs involving no-cost exploitations. It cannot detect more complex bugs where attackers incur some costs for disproportionate gains. Addressing these sophisticated scenarios requires a deeper understanding of DeFi's business logic, which we will explore in future research.

References

[1] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2009. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[2] "Cryptocurrency marketcap," 2024, accessed Aug, 2024. [Online]. Available: https://coinmarketcap.com/

[3] "DefiLlama," 2023, accessed July, 2023. [Online]. Available: https://defillama.com/

[4] C. Team, "2022 Biggest Year Ever For Crypto Hacking," Feb. 2023. [Online]. Available: https://www.chainalysis.com/blog/2022-biggest-year-ever-for-crypto-hacking/

[5] "Understanding The DAO Attack," Jun. 2016, section: Learn. [Online]. Available: https://www.coindesk.com/learn/understanding-the-dao-attack/

[6] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "SAILFISH: Vetting Smart Contract State-Inconsistency Bugs in Seconds," Dec. 2021, arXiv:2104.08638 [cs]. [Online]. Available: http://arxiv.org/abs/2104.08638

[7] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 1186–1189.

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

[9] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying Exploitable Bugs in Smart Contracts." ICSE, 2023.

[10] "Token Standards," 2023. [Online]. Available: https://ethereum.org

[11] S. Finance, "Swivel Finance." [Online]. Available: https://swivel.finance

[12] "Insure - Code4rena," 2022. [Online]. Available: https://code4rena.com/reports/2022-01-insure

[13] "Hardhat," 2023. [Online]. Available: https://hardhat.org

[14] "Truffle Suite," 2023. [Online]. Available: https://trufflesuite.com/

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

[16] "OpenZeppelin," 2015. [Online]. Available: https://www.openzeppelin.com/

[17] "NLTK :: Natural Language Toolkit," 2001. [Online]. Available: https://www.nltk.org/

[18] "Code4rena," 2021. [Online]. Available: https://code4rena.com/

[19] "Level bug." [Online]. Available: https://rekt.news/level-finance-rekt/

[20] "SUPER SUSHI SAMURAI bug." [Online]. Available: https://x.com/coffeexcoin/status/1770834359601217886

[21] Chaofan Shou [@shoucccc], "MINER bug," Feb. 2024. [Online]. Available: https://x.com/shoucccc/status/1757777764646859121

[22] PeckShield Inc. [@peckshield], "GYMNET bug," Jun. 2022. [Online]. Available: https://x.com/peckshield/status/1534423219607719936

[23] Ishu [@ishwinder], "Meter bug," Feb. 2022. [Online]. Available: https://x.com/ishwinder/status/1490227406824685569

[24] "Qbridge bug." [Online]. Available: https://medium.com/@QubitFin/protocol-exploit-report-2-30aade4d66de

[25] "Socket bug." [Online]. Available: https://www.rekt.news/

[26] "Omm exploit postmortem | January 2023," Jan. 2023. [Online]. Available: https://forum.omm.finance/t/omm-exploit-postmortem-january-2023/266

[27] PeckShield Inc. [@peckshield], "Rubic bug," Dec. 2022. [Online]. Available: https://x.com/peckshield/status/1606942321115033600

[28] "Transit Swap bug." [Online]. Available: https://www.rekt.news/

[29] etherscan.io, "Ethereum (ETH) Blockchain Explorer." [Online]. Available: https://etherscan.io/

[30] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. Vienna Austria: ACM, Oct. 2016, pp. 254–269. [Online]. Available: https://dl.acm.org/doi/10.1145/2976749.2978309

[31] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2017, pp. 442–446.

[32] C. F. Torres, J. Schütte, and R. State, "Osiris: Hunting for integer bugs in ethereum smart contracts," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 664–676.

[33] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *Proceedings of the 34th annual computer security applications conference*, 2018, pp. 653–663.

[34] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *27th ${$USENIX$}$ Security Symposium (${$USENIX$}$ Security 18)*, 2018, pp. 1317–1333.

[35] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "sCompile: Critical path identification and analysis for smart contracts," in *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019, Proceedings 21*. Springer, 2019, pp. 286–304.

[36] S. So, S. Hong, and H. Oh, "SmarTest: Effectively Hunting Vulnerable Transaction Sequences in Smart Contracts through Language Model-Guided Symbolic Execution." in *USENIX Security Symposium*, 2021, pp. 1361–1378.

[37] Z. Wang, B. Wen, Z. Luo, and S. Liu, "Mar: A dynamic symbol execution detection method for smart contract reentry vulnerability," in *Blockchain and Trustworthy Systems: Third International Conference, BlockSys 2021, Guangzhou, China, August 5–6, 2021, Revised Selected Papers 3*. Springer, 2021, pp. 418–429.

[38] "Mythril classic: an open-source security analysis tool for ethereum smart contracts," Aug. 2023, original-date: 2017-09-18T04:14:20Z. [Online]. Available: https://github.com/Consensys/mythril

[39] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018, pp. 65–68.

[40] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 259–269.

[41] J. He, M. Balunović, N. Ambroladze, P. Tsankov, and M. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 531–548.

[42] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: An efficient adaptive fuzzer for solidity smart contracts," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 778–788.

[43] V. Wüstholz and M. Christakis, "Harvey: A greybox fuzzer for smart contracts," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1398–1409.

[44] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: catching vulnerable smart contracts once and for all," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2019, pp. 1–4.

[45] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 3, pp. 1795–1809, 2020, publisher: IEEE.

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

[47] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "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, 2021, pp. 227–239.

[48] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 557–560.

[49] A. Groce and G. Grieco, "Echidna-parade: A tool for diverse multicore smart contract fuzzing," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 658–661.

[50] Y. Xue, J. Ye, W. Zhang, J. Sun, L. Ma, H. Wang, and J. Zhao, "xfuzz: Machine learning guided cross-contract fuzzing," *IEEE Transactions on Dependable and Secure Computing*, 2022, publisher: IEEE.

[51] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, pp. 1–28, 2017, publisher: ACM New York, NY, USA.

[52] Hajdu and D. Jovanović, "solc-verify: A modular verifier for solidity smart contracts," in *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*. Springer, 2020, pp. 161–179.

[53] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York*

City, NY, USA, July 13–14, 2019, Revised Selected Papers 11. Springer, 2020, pp. 87–106.

[54] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VeriSmart: A highly precise safety verifier for Ethereum smart contracts," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1678–1694.

[55] B. Tan, B. Mariano, S. Lahiri, I. Dillig, and Y. Feng, "Soltype: Refinement types for solidity," *arXiv preprint arXiv:2110.00677*, 2021.

[56] Y. Liu, Y. Li, S.-W. Lin, and R. Zhao, "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*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 666–677. [Online]. Available: https://doi.org/10.1145/3368089.3409740

[57] B. Zhang, "Towards finding accounting errors in smart contracts," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[58] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," *arXiv preprint arXiv:1812.05934*, 2018.

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

[60] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *Proceedings of the 1st international workshop on emerging trends in software engineering for blockchain*, 2018, pp. 9–16.

[61] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *Ndss*, 2018, pp. 1–12.

[62] Y. Sun, D. Wu, Y. Xue, H. Liu, H. Wang, Z. Xu, X. Xie, and Y. Liu, "Gptscan: Detecting logic vulnerabilities in smart contracts by combining gpt with program analysis," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.