Safeguarding DeFi Smart Contracts against Oracle Deviations

Xun Deng
xun.deng@mail.utoronto.ca
University of Toronto
Toronto, Canada

Sidi Mohamed Beillahi
sm.beillahi@utoronto.ca
University of Toronto
Toronto, Canada

Cyrus Minwalla
cminwalla@bank-banque-canada.ca
Bank of Canada
Ottawa, Canada

Han Du
HDu@bank-banque-canada.ca
Bank of Canada
Ottawa, Canada

Andreas Veneris
veneris@eecg.toronto.edu
University of Toronto
Toronto, Canada

Fan Long
fanl@cs.toronto.edu
University of Toronto
Toronto, Canada

ABSTRACT

This paper presents OVer, a framework designed to automatically analyze the behavior of decentralized finance (DeFi) protocols when subjected to a "skewed" oracle input. OVer firstly performs symbolic analysis on the given contract and constructs a model of constraints. Then, the framework leverages an SMT solver to identify parameters that allow its secure operation. Furthermore, guard statements may be generated for smart contracts that may use the oracle values, thus effectively preventing oracle manipulation attacks. Empirical results show that OVer can successfully analyze all 10 benchmarks collected, which encompass a diverse range of DeFi protocols. Additionally, this paper illustrates that current parameters utilized in the majority of benchmarks are inadequate to ensure safety when confronted with significant oracle deviations. It shows that existing ad-hoc control mechanisms such as introducing delays are often insufficient or even detrimental to protect the DeFi protocols against the oracle deviation in the real-world.

CCS CONCEPTS

• Software and its engineering → Formal software verification; **Designing software**; **• Security and privacy → Software security engineering**.

KEYWORDS

Blockchain, Decentralized Finance, Smart Contracts, Oracle Deviation, Static Program Analysis, Code Summary, Parameter Optimization

ACM Reference Format:
Xun Deng, Sidi Mohamed Beillahi, Cyrus Minwalla, Han Du, Andreas Veneris, and Fan Long. 2024. Safeguarding DeFi Smart Contracts against Oracle Deviations. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639225

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04
https://doi.org/10.1145/3597503.3639225

1 INTRODUCTION

Blockchain offers decentralized, programmable, and robust ledgers on a global scale. Smart contracts, which are programs deployed on blockchains, encode transaction rules to govern these blockchain ledgers. This technology has been adopted across a wide range of sectors, including financial services, supply chain management, and entertainment. A notable application of smart contracts is in the management of digital assets to create decentralized financial services (DeFi). As of April 1st, 2023, the Total Value Locked (TVL) in 1,417 DeFi contracts had reached $50.15 billion [17].

As the assets managed by smart contracts continue to grow, ensuring their correctness has become a critical issue. In response, researchers have developed numerous analysis and verification tools to detect errors in contract implementation. However, beyond the typical software challenges posed by implementation errors, the correctness of many DeFi smart contracts often depends on *oracle values* [4]. These are external values that capture vital environmental conditions under which the contracts operate. For instance, a collateralized DeFi lending contract requires updated trading prices of various digital assets to ensure that the value of the collateral asset always exceeds the value of the borrowed asset for each user.

Smart contracts periodically receive updates to their oracle values from other contracts or external databases and APIs. Deviations in these oracle values from their true values can lead to deviations in the intended operations of the contracts [4, 9]. In the real world, such deviations are common, often stemming from inaccuracies in the value source or delays in transmission. DeFi protocols traditionally use a variety of empirical strategies to mitigate the risks associated with oracle deviations and potential corruptions. For instance, a leveraged DeFi protocol might set a safety margin for user positions, liquidating a position if its asset price dips below a specific threshold. Alternatively, a protocol might aggregate multiple oracle inputs from varied sources, calculating a median or average for computational purposes. However, these mechanisms and their parameters are often ad-hoc and arbitrary. The adequacy and efficacy of these control mechanisms in real-world scenarios remain uncertain.

This paper presents OVer, the first sound and automated tool for analyzing oracle deviation and verifying its impact in DeFi smart contracts. Given the source code of a smart contract protocol and a deviation range of specific oracle values in the contracts, OVer automatically analyzes the source code to extract a summary of the protocol. For a safety constraint of the protocol, OVer then uses the extracted summary to determine how to appropriately set key control parameters in the contract. This ensures that the resulting contract continues to satisfy the desired constraint, even in the face of oracle deviations.

One of the key challenges OVer faces is the sophisticated contract logic of DeFi protocols. DeFi contracts often contain multiple loops that iterate over map-like data structures. Each iteration typically contains up to a hundred lines of code to handle the protocol logic for one kind of asset or for one user account. Such code patterns are typically intractable for standard program analysis techniques, which often would have to make undesirable over-approximation or to bound the number of loop iterations, leading to inaccurate or unsound analysis results.

OVer tackles this challenge with its innovative loop summary algorithm. Since the essence of loops computations in DeFi protocols consists of accumulators applied to maps data structures, OVer operates with a predefined sum operator template for loops. OVer extracts the summary formula of each iteration and then uses a template-based approach to convert the extracted expressions into an instantiation of the sum operator template to represent the summary of the entire loop. Distinct from previous loop summary algorithms that struggle with complex if-else branching or multi-faceted folding operations with interdependencies [37], the OVer algorithm adeptly manages these prevalent complexities in popular DeFi contracts.

We have evaluated OVer on a set of nine popular DeFi protocols and one fictional protocol in our experiments. OVer successfully analyzes all the protocols, each taking less than nine seconds. In comparison, a prior state-of-the-art loop summary algorithm can only handle 0 out of 7 benchmarks that have loops.

With OVer, we study the history oracle deviation in real-world blockchains. We investigate how oracle deviation would affect the behavior of popular DeFi contracts and whether the existing ad-hoc mechanisms are sufficient to neutralize the oracle deviations. Our results show that for six out of the seven benchmark protocols, the control mechanism was insufficient to handle the oracle deviation for at least a certain period of time, leading to temporary exploitable vulnerabilities. Our results also surprisingly show that existing ad-hoc mechanisms often exacerbate the security issue caused by oracle deviation. For example, to protect against potentially malicious oracle value providers, several DeFi protocols introduce delays when using oracle value inputs in their calculation (*e.g.*, using the reported asset price one hour ago as the current oracle price). When the digital asset price fluctuates, such mechanisms fail to reflect the current market and artificially inject deviations, which may make the resulting protocols more vulnerable.

In summary, this paper makes the following contributions.

- *OVer:* This paper presents OVer, the first sound analysis and verification tool for analyzing oracle deviation in DeFi protocols.
- *Loop Summary Algorithm:* This paper proposes a novel loop summary algorithm to enable the analysis of the sophisticated loops in DeFi smart contract source code.
- *Results:* This paper presents a systematic evaluation of OVer. It also presents the first study of oracle deviation on popular DeFi protocols. Our results show that the existing ad-hoc control mechanisms are often insufficient or even detrimental to protect the DeFi protocols against the oracle deviation in the real-world.

The remaining of the paper is organized as follows. Sections 2-3 introduce technical background and a motivating example. Section 4 presents the design of OVer. In Section 5, we study past oracle deviations and evaluate OVer. We discuss related work and threats to the validity in Sections 6-7 and conclude in Section 8.

2 BACKGROUND

Blockchain and smart contracts. Blockchains, operating as decentralized distributed systems, offer a formidable architecture for resilient, programmable ledgers. Numerous blockchain infrastructures, with Ethereum as a prime example, provide support for smart contracts. These are coded agreements residing on the blockchain, established to administer transaction rules integral to ledger operations. Commonly scripted in sophisticated languages such as *Solidity* [29], these smart contracts are later compiled into a lower-level machine language like Ethereum Virtual Machine (EVM) bytecode [8]. For consistent enforcement of these transaction rules, all participating nodes within the blockchain network execute the bytecode of a contract in a consensus-oriented fashion.

Decentralized finance protocols. A substantial application of blockchain technology is visible in the form of DeFi protocols. DeFi protocols deploy smart contracts to manage digital assets, enabling an array of financial services encompassing trading, lending, and investment, all within a decentralized context. Predominantly, DeFi applications consist of automatic market makers (AMMs) and lending protocols, with AMMs being a frequent component of decentralized exchanges (DEXes).

Contrasting traditional exchanges that utilize order books for trading operations, AMMs implement a mathematical model that is contingent on the asset's volume in the liquidity pool to ascertain an asset's price. Furthermore, a majority of DeFi lending protocols mandate borrowers to provide over-collateralization, instigating liquidation if a borrower's position descends to under-collateralization. To maintain functional efficiency, lending protocols integrate key parameters such as collateralization or liquidation ratios.

Blockchain oracles. Oracles provide real-world data to blockchains as they are tightly-closed systems and agnostic to such information. Thus, oracles are critical for the smooth operation of DeFi protocols. Specifically, price oracles furnish indispensable information that has direct implications on both smart contract execution and their results. For instance, lending protocols use exact collateral asset prices to gauge user risk profiles, and outdated or imprecise data may precipitate financial losses.

In relation to oracle inputs, two distinct types of deviations can occur: *accuracy* and *latency*. Accuracy deviations emerge when a value deviates from its actual or true value, while latency deviations are identified when outdated values are reported, a phenomenon that can, in turn, influence accuracy. These deviations can originate from various sources such as intentional manipulation of oracles to report distorted values, or unintentional data adjustments embedded within smart contracts. Irrespective of their origins, such deviations can result in incorrect operations within smart contracts.

```
1  function borrowAllowed(address cToken, address bwr, uint brwAmt) external returns (uint) {
2    ...
3    uint surplus = hypotheticalLiquid(bwr,cToken,0,brwAmt);
4    require(surplus > 0, "INSUFFICIENT_LIQUIDITY");
5    ...
6    return uint(Error.NO_ERROR);   }
7
8  function hypotheticalLiquid(address acct, CToken cToken, uint redTok, uint brwAmt) internal returns (uint) {
9    AccountLiquidityLocalVars memory v;
10   // Iterate over each asset in the acct
11   CToken[] memory assets = accountAssets[acct];
12   for (uint i = 0; i < assets.length; i++) {
13     CToken asset = assets[i];
14     (, v.cTokenBal, v.brwBal, v.exchRt) =
15     asset.getAccountSnapshot(acct);
16     // Fetch asset price from oracle
17     v.oraclePrice = oracle.getUnderlyingPrice(asset);
18     v.collFact = markets[address(asset)].collFact;
19     v.tokensToDenom= v.collFact*v.exchRt*v.oraclePrice;
20     v.sumColl = v.sumColl+ v.tokensToDenom* v.cTokenBal;
21     v.sumBrwEfct= v.sumBrwEfct+ v.oraclePrice* v.brwBal;
22     if (asset == cToken) {
23       v.sumBrwEfct= v.sumBrwEfct+ v.tokensToDenom*redTok;
24       v.sumBrwEfct= v.sumBrwEfct+ v.oraclePrice*brwAmt;}}
25   return v.sumColl - v.sumBrwEfct;                    }
```

Figure 1: *Compound* protocol borrow logic simplified.

Complexity of DeFi smart contracts. Smart contracts implementing DeFi protocols such as lending, DEXes, and derivatives [1, 3, 26] can be complex since they generally include loops to iterate through data structures representing various assets types or accounts managed by the protocols and calculate a sum, e.g., total assets or debts. This work highlights that a typical *Solidity* contract tends to include one loop per 250 lines of cods and over 60% of loops perform an accumulation [37].

3 EXAMPLE AND OVERVIEW

We present a motivating example of applying OVer to analyze oracle deviation in *Compound* [13]. Figure 1 presents a simplified code snippet from the *Compound* smart contracts. *Compound* is a decentralized borrowing and lending protocol operating on the Ethereum blockchain. To borrow assets from *Compound*, a user deposits assets as collateral. The total value of the collateral has to be significantly greater than the value of the borrowed assets at any time. Whenever a user attempts to borrow assets, *Compound* calls `borrowAllowed` (line 1 in Figure 1) to enforce this policy. The function `borrowAllowed` in turn calls `hypotheticalLiquid` (line 8) to calculate the difference (*i.e.*, `surplus` at line 4) between the adjusted value of the collateral assets (*i.e.*, `v.sumColl` at line 20) and the total value of the borrowed assets (*i.e.*, `v.sumBrwEfct` at line 21) for the given account `acct`. In the function, *Compound* computes these two values with the loop at lines 12-24. Each iteration of the loop handles one kind of asset in *Compound* and updates the two variables. Specifically, the loop computes `v.sumColl` as follows:

$$\sum_{a\in assets}(collFact_a * exchRt_a * p_a * cTokenBal_a) \quad (1)$$

where $exchRt_a$ is the exchange rate of the collateral asset a, p_a is the price of the asset a fetched from an external oracle contract (`v.oraclePrice` at line 20), $cTokenBal_a$ is the balance of the asset a, and $collFact_a$ is a control variable smaller than one to determine the enforced over-collateralization ratio for the asset. The loop also computes `v.sumBrwEfct` as follows:

$$\sum_{a\in assets}(brwBal_a*p_a+c_a*(p_{brw}*brwAmt+collFact_r*exchRt_r*p_r*redTok)) \quad (2)$$

where p_{brw} is the price of the asset brw that the user wants to borrow, r is the asset the user wants to withdraw from its collateral, $brwBal_a$ is the already borrowed balance of the asset a, $brwAmt$ is the amount of the asset brw a user wants to borrow, and $redTok$ is the amount of asset r a user wants to redeem. $c_a = Int(a == cToken)$ is a binary representation of the condition at line 22, where $c_a = 1$ when the asset a is $cToken$ and $c_a = 0$ otherwise. Because `hypotheticalLiquid` can be invoked when a user borrows assets or redeems collaterals, there are two different cases. The second term corresponds to the borrowing case, while the last term corresponds to the redeem case.

Oracle values in Compound. The correctness of *Compound* depends on the accuracy of the fetched oracle price of each asset (line 17). Like many other DeFi protocols, *Compound* fetches oracle prices from multiple sources, including centralized oracle service providers such as *Chainlink* [28] and the trading price of the assets in decentralized protocols such as *Uniswap* [32]. However, values from these sources may deviate from ground truths. In fact, when a digital asset price is volatile, obtaining the fair price of an asset is typically impossible. For instance, if the prices in the equation 1 are inaccurately reported as high, the value of users' collateral would increase, potentially leading the protocol to execute borrowing transactions even when users are not sufficiently collateralized.

To tackle this issue, *Compound* enforces additional margins for the positions of each collateral asset and the margin sizes are determined by `collFact`. *Compound* empirically sets the collateral factor value lower to enforce a larger margin on more volatile assets and sets the factor higher on less volatile assets. Many other DeFi protocols have similar ad-hoc control mechanisms to protect against oracle deviations. But there is a difficult trade-off in how to set these control parameters appropriately. On one hand, setting the parameters too relaxed would make the contracts vulnerable when facing oracle deviations. On the other hand, setting the parameters too restrictive would place additional collateral burden on users and make the protocol unattractive.

Utilizing OVer. We now show how we apply OVer to analyze *Compound* to determine optimal control parameter values such as the collateral factor. The user first identifies the interested operations in the source code. In our example, we identify `borrowAllowed` as entry point and `hypotheticalLiquid`, which does critical checks and computations when performing borrowing actions.

Code analysis. OVer first analyzes the source code in Figure 1 to generate a symbolic expression for all variables in the constraint at line 4. It starts with the entry function, replacing intermediate variables with their computed expressions. For example, `surplus` is the returned value of `hypotheticalLiquid`. It is computed by subtracting `v.sumBrwEfct` from `v.sumColl`. The expressions extracted by OVer for `v.sumColl` and `v.sumBrwEfct` correspond to the mathematical formulas in Equations 1 and 2, respectively.

OVer then generates the final symbolic expression for the safety constraint at line 4 in Figure 2. Note that the terms in the final

$\sum_{a \in assets}(collFact_a * exchRt_a * p_a * cTokenBal_a) - (\sum_{a \in assets}(brwBal_a * p_a + c_a * (p_{brw} * brwAmt + collFact_r * exchRt_r * p_r))) > 0$

Figure 2: Compound analysis summary.

expression are either loaded contract states (*e.g.*, `v.brwBal`) or the return values of external function calls (*e.g.*, `getUnderlyingPrice`).

Loop summerization. OVer handles the loop at lines 12-26 as follows. With the observation that most loops in DeFi contracts perform fold operations, particularly accumulation, OVer summarizes the loop by identifying all accumulation performed and replacing the loop with one or multiple compact expression(s). By replacing the variables and loops with compact expressions, the code summary module returns a set of constraints to represent the smart contract's logic. Constraints that are not affected by oracles will be ignored. In this example, the constraint at line 4 in Figure 1 will be extracted as the summary shown in Figure 2. Note that, in this summary, there are five vector variables and three scalar variables.

Formal model generation. The analysis results in Figure 2 are then used to construct a sound model of the safety constraint. Suppose we want to investigate the behavior of the borrowing function in *Compound* and identify the price deviation limit when using the default collateral factor (cf) 0.7, and a target collateral factor (cf') 0.75. Note that because of the deviation, the target value is always greater than the one configured in the contract. From the expression in Figure 2, we can derive the following simplified model:

$$\min_{\delta} \ \delta$$
$$\text{s.t. } \forall\, C, D, b, P, p, P_b, p_b > 0, \frac{|P_i - p_i|}{P_i} < \delta, \frac{|P_b - p_b|}{P_b} < \delta,$$
$$cf * \sum_{i}^{len}(C_i - D_i) * p_i - p_b * b > 0 \Rightarrow cf' * \sum_{i}^{len}(C_i - D_i) * P_i - P_b * b > 0$$

In this model, the variables C, D, b represent *CollBal*, *brwBal* and *brwAmt* respectively. P and P_b stand for ground truth values while p and p_b stand for values reported by the oracle. Note that because we are analyzing the borrowing case, the redeem amount is always zero and therefore the redeem related terms are simplified away.

Formal SMT solution. Finally, we pass the model to the optimizer, which iteratively calls an SMT solver to prove the constraint specified in the model above. Following, it returns the optimal δ if one is found. Then we can insert proper `require` statements, for instance, restricting oracle deviation to be less than the value found, into the source code to ensure correct behavior.

4 DESIGN

First, we introduce a simplified *Solidity* language to help present our proposed analysis framework. The language, shown in Listing 1, captures variable declarations, assignments, control-flow structures, and function calls.

Contract, State, and Function. A contract class has an identifier (id) of String type (line 1 in Listing 1). It encompasses a set of global states and functions. Each global state has a type and an identifier. We specify a function with its name, parameters and body which is constituted of a sequence of statements. The statements *dec* and *assign* (line 8) allow to declare a local variable and assign value to it, respectively. The statement *load* allows to read a contract's state.

Control-Flow Structures. Conditional branches are represented by *IfThenElse* construct. A *for* loop is constituted of the loop iterator i,

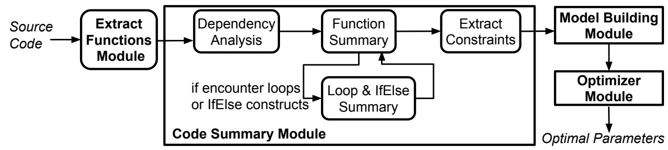

Figure 3: Overview of the proposed framework.

upper-bound n (for simplicity we omitted lower-bound), and a sequence of statements representing the loop body. A *phi* instruction, denoted as ϕ_{id} is used to select values based on the control flow for a static single-assignment (SSA) smart contract. It can only appear at the beginning in a loop body or after an *IfThenElse* construct. The *require* statement enforces smart contract constraints and its logic, and is crucial in our analysis.

Function Calls. Function calls are represented by the statement *call(*f, $\mathcal{E}^*$, id^**)*, where f identifies the function, $\mathcal{E}^*$ denotes the set of parameters, and id^* represents the names of return values. Some function calls represent queries of states and to oracle. In our optimization problem, we consider those as free variables. Table 1 shows examples of statements from the *Compound* protocol.

```
1  Contract C ::= contract(id,St*,F*)
2  State st ::= state(ty,id)
3  Func F ::= func(f,id*,S*)
4  Type ty  ::= Int|Bool|Struct|Map|Array|Bytes|Address
5  Id   id  ::= String| f,i ∈ Id
6  Const c  ::= n ∈ Int|Bool|Bytes|Address
7  Op ::= + | - | × | / | >= | > | < | <= | =
8  Stmt S::= dec(ty,id) | assign(id,E) |
9            load(id,st) | require(E) |
10           phi(id_0,id_1,...) |
11           if(E_c)then{S_1*}else{S_2*}Φ* |
12           for(i,n){S*} |
13           call(f,E*,id*) | return(E*)
14 Expr E ::= c | id | id[E] | id.id | ¬E | E_1 Op E_2
```

Listing 1: Simplified Solidity language.

4.1 Code Summary Overview

Figure 3 presents an overview of the proposed framework. The high-level procedure for the framework is shown in Algorithm 1. The first step is to preprocess and simplify the source code to SSA form using the module *ExtractFunc*. In *ExtractFunc*, we identify the entry point of our analysis. Note that the entry point function must be a public function. In the case of *Compound*, this entry-point is the *borrowAllowed* function. Furthermore, the extracted functions are *pure* functions, and they are invoked through the *call* mechanism[1]. Then, *CodeSummary* module extracts concise summaries including summaries of loops and conditionals and returns a list of constraints. Next, *BuildModel* module constructs an optimization model from the list of constraints for an SMT solver. Lastly, *SolveOpt* module solves the optimization model using the SMT solver.

[1]In our implementation, we parse the abstract syntax tree (AST) of smart contracts and perform analysis on nodes in the extracted AST tree to identify pure functions. Pure functions are functions that have no side effects, *i.e.*, do not modify contract's global states.

Table 1: Example statements from Compound smart contracts in Solidity and simplified Solidity.

Solidity Code	require(surplus > 0, "INSUFFICIENT_LIQUIDITY")	v.oraclePrice = oracle.getUnderlyingPrice(asset)
Simplified Solidity	require(surplus > 0)	call(oracle.getUnderlyingPrice, asset, v.oraclePrice)

Algorithm 1 *Main* procedure. It takes in source code *sc* of the benchmark.

```
procedure SummaryAnalysis(sc)
    FuncObj ← ExtractFunc(sc)
    ConstLst ← CodeSummary(FuncObj)
    M ← BuildModel(ConstLst)
    OptVar ← SolveOpt(M)
    return OptVar
```

4.2 Code Summary Module

The main tasks of the code summary module are: loop and oracle dependencies analysis, extraction of symbolic expressions for variables, and constraints extraction. To compute a concise symbolic expression that can be passed to a solver, we summarize loops' bodies using the accumulation operator. Towards our goal, we introduce a domain-specific language (DSL) shown in Listing 2.

```
aop      ::= +, -, *, /    ;   bop      ::= >, >=, <, <=, =
Id, i    ::= String        ;   lb, ub  ::= Int
Const    ::= Int | Bool | Bytes | Address
Val 𝕍    ::= Id | Const | i | index(𝕍₁,𝕍₂) | 𝕍₁(𝕍₂) |
             𝕍₁ aop 𝕍₂ | ret(𝕍₂, i)
Acc      ::= sum(𝔼, i, ub)
Expr 𝔼   ::= Acc | 𝕍 | 𝔼 aop 𝔼
Constr ℂ::= True | 𝔼 bop 𝔼 | ¬ℂ
```

Listing 2: Code summary DSL.

The two main components of our DSL are $\mathbb{E}$ for expressions and $\mathbb{C}$ for Boolean constraint expressions. The $\mathbb{E}$ type can either be an accumulation value (*Acc*), a value ($\mathbb{V}$), or an arithmetic operation between two expressions. The $\mathbb{C}$ type can either be the constant Boolean value *True*, a comparison operation between two expressions, or a negation of another constraint.

Indexing and member-access. We use the index operator *index(*$\mathbb{V}_1$, $\mathbb{V}_2$*)* to represent accessing an element from the array or map $\mathbb{V}_1$ with the key $\mathbb{V}_2$. The type of $\mathbb{V}_1$ must be either an array or map and the type of $\mathbb{V}_2$ must be the same as the key's type of $\mathbb{V}_1$. This definition of the index operator allows nested indexing. The member-access operator $\mathbb{V}_1(\mathbb{V}_2)$ is used to represent accessing the field $\mathbb{V}_1$ of the struct variable $\mathbb{V}_2$.

Accumulation value. To represent a loop's summary in our DSL, we use the accumulation operator *sum(*$\mathbb{E}$*, i, ub)*, where *i* is the iterator and *ub* is the upper bound. The complexity of the summation is captured in the term $\mathbb{E}$, where it can be a complex mathematics formula involving multiple index and member-access operators.

Return values of function calls. We utilize *ret(*$\mathbb{V}$*, i)* to indicate that $\mathbb{V}$ is the return value of a *pure* function which reads global states. When $\mathbb{V}$ is a loop-dependent, *i* represents the loop iterator, otherwise, *i* is *null*. For example, at line 18 in Figure 1, *v.oraclePrice* gets the value from the function *oracle.getUnderlyingPrice* and the function is loop-dependent because of the argument *asset*. The generated summary is *ret(oraclePrice(v), i)*.

4.2.1 Dependency Analysis. To determine expressions and require statements to include in our optimization model, we need to find which variables depend on the oracle price. Towards this, we propose a set of rules *O1-O4* to infer this dependency and introduce the rule *O5* to find guard statements that are oracle dependent. Moreover, to compute the loop summary we need to find which variables inside a loop body that depends on the loop iterator in order to account for them in the accumulation operator of our DSL. Thus, we also propose the set of rules *L1-L4* to infer loop dependency.

OD denotes the set of expressions and statements that are oracle dependent. We do not distinguish between the two in *OD*. The union operation for sets is denoted as $\uplus$.

O1: If pure function reads from an oracle state then the identifier it reads to is oracle dependent. If $\mathcal{S}$ *:= call(f,* $\mathcal{E}$*, id)* and *Identifier(f) = oracle* where the helper function *Identifier* checks whether the function is annotated as an oracle state getter, then $id \in OD$.

$$\frac{\mathcal{S} := call(f, \mathcal{E}, id), Identifier(f) = oracle}{OD = OD \uplus \{id,\ \mathcal{S}\}}$$

O2: If a statement reads from a global state that is oracle dependent and if a statement assigns an expression that is oracle dependent to a variable then the variable is oracle dependent.

$$\frac{\mathcal{S} := load(id, st), st \in OD}{OD = OD \uplus \{id, \mathcal{S}\}} \quad \frac{\mathcal{S} := assign(id, \mathcal{E}), \mathcal{E} \in OD}{OD = OD \uplus \{id,\ \mathcal{S}\}}$$

O3: For arithmetic and comparison expressions, if one of the operands is oracle dependent then the result is oracle dependent.

$$\frac{\mathcal{E} = \mathcal{E}_1\ op\ \mathcal{E}_2, \mathcal{E}_1 \in OD \vee \mathcal{E}_2 \in OD}{OD = OD \uplus \{\mathcal{E}\}}$$

O4: For a function *f*, if its parameter $\mathcal{E}$ or one of its statements is oracle dependent, then the return value *id* is oracle dependent.

$$\frac{\mathcal{S} := call(f, \mathcal{E}, id), f := func(f,, \mathcal{S}'^*), \mathcal{E} \in OD \vee \mathcal{S}'^* \in OD}{OD = OD \uplus \{id,\ \mathcal{S}\}}$$

O5: A require statement is oracle dependent if its expression is.

$$\frac{\mathcal{S} := require(\mathcal{E}), \mathcal{E} \in OD}{OD = OD \uplus \{\mathcal{S}\}}$$

We use L_i to denote a `for` loop, with iterator *i*, and corresponds to $\mathcal{S} = for(i, n, \mathcal{S}^*)$. LD_i is the set of expressions that depend on L_i.

L1: If an expression with an index that corresponds to the loop iterator or is loop dependent then the expression is loop dependent.

$$\frac{\mathcal{E} = id[\mathcal{E}], \mathcal{E} = i \vee \mathcal{E} \in LD_i}{LD_i = LD_i \uplus \{\mathcal{E}\}}$$

L2: For arithmetic and comparison expressions, if one of the operands is loop dependent then the result is loop dependent.

$$\frac{\mathcal{E} = \mathcal{E}_1\ op\ \mathcal{E}_2, \mathcal{E}_1 \in LD_i \vee \mathcal{E}_2 \in LD_i}{LD_i = LD_i \uplus \{\mathcal{E}\}}$$

L3: If a statement assigns an expression to a variable and this expression is loop dependent then the variable is loop dependent.

$$\frac{\mathcal{S} := assign(id, \mathcal{E}), \mathcal{E} \in LD_i}{LD_i = LD_i \uplus \{id\}}$$

$$\frac{pc(f) := \text{require } \mathcal{E} \textbf{ or } pc(f) := \text{return } \mathcal{E}, \mathbb{S}(f) = \perp, \mathbb{E} := ConvDSL(\mathcal{E})}{\mathbb{S} = \mathbb{S}[f \mapsto \mathbb{E}]}$$

$$\frac{pc(f) := assign(id, \mathcal{E}), \mathbb{S}(f) = \{\mathcal{E}\}, \mathbb{E}' := \mathbb{E}[id/ConvDSL(\mathcal{E})]}{\mathbb{S} = \mathbb{S}[f \mapsto \mathbb{E}']}$$

$$\frac{pc(f) := call(f', \mathcal{E}, id), \mathbb{S}(f) = \mathbb{E}, \mathbb{E}' := \mathbb{E}[id/\mathbb{S}(f'(\mathcal{E}))]}{\mathbb{S} = \mathbb{S}[f \mapsto \mathbb{E}']}$$

$$\frac{pc(f) := for(i, n)\{\mathcal{S}^*\}, \mathbb{E}' := LpSm(i, n, \mathcal{S}^*, \mathbb{S}(f))}{\mathbb{S} = \mathbb{S}[f \mapsto \mathbb{E}']}$$

$$\frac{pc(f) := if(\mathcal{E}_c)then\{\mathcal{S}_1^*\}else\{\mathcal{S}_2^*\}, \mathbb{E}' := IfSm(\mathcal{E}_c, \mathcal{S}_1^*, \mathcal{S}_2^*, \Phi^*, \mathbb{S}(f))}{\mathbb{S} = \mathbb{S}[f \mapsto \mathbb{E}']}$$

Figure 4: Function summary extraction rules.

L4: If a statement invokes a function f with a parameter $\mathcal{E}$ that is loop dependent, then the return value id is loop dependent.

$$\frac{\mathcal{S} := call(f, \mathcal{E}, id), \mathcal{E} \in LD_i}{LD_i = LD_i \uplus \{id\}}$$

4.2.2 Symbolic Value Extraction. We now present a set of rules to generate a function summary, shown in Figure 4. We use *ExtractSummary* to refer to those rules. Specifically, *ExtractSummary* takes a statement $\mathcal{S}$ and an expression $\mathcal{E}$. It applies the effect of $\mathcal{S}$ on $\mathcal{E}$ and returns the updated expression $\mathcal{E}'$. We use $\mathbb{S}$ that maps a function f to its summary. Our approach uses a *bottom-up* algorithm that begins from a *return* or a *require* statements. It adds the return expression $\mathcal{E}$ of a function f to the initially empty summary $\mathbb{S}(f)$.

For an assignment, $id = \mathcal{E}$, we convert $\mathcal{E}$ to its DSL using the procedure *ConvDSL*. We substitute all occurrences of *id* in $\mathbb{S}(f) = \mathbb{E}$ with its computed DSL value, denoted as $\mathbb{E}[id/ConvDSL(\mathcal{E})]$.

For a function call, *call(f', $\mathcal{E}$, id)*, we generate a summary of *f'($\mathcal{E}$)*, denoted as *S(f'($\mathcal{E}$))*, and replace symbol *id* with *S(f'($\mathcal{E}$))*.

The more involved summarization of loops and if-else statements are handled in the *LpSm* and *IfSm* procedures that we describe next.

Loop Summary. In Algorithm 2, we present the procedure *LpSm*. *LpSm* attempts to generate summaries for the symbols in the expression $\mathbb{E}$, in the form of accumulation or nested accumulation. For each symbol, we analyze the *for* loop body in a bottom-up manner and perform symbolic substitution using *ExtractSummary*. We enumerate the symbols following their order of dependency (line 4). For instance, in Listing 3, since *acc1* depends on *acc* we enumerate *acc1* before *acc*. We use *ps* to denote the "partial summary" of the symbol's value at an iteration *i*.

```
acc: acc_0 + sum(index(A, j),j,b)
acc1: acc1_0 + sum(acc_0 + sum(index(A, j),j,k),k,b)
for (i =  0; i < b; i ++)  {
    acc' = phi(acc_0, acc_{i-1})
    acc1' = phi(acc1_0, acc1_{i-1})
    acc_i =  acc' + A[i]
    acc1_i = acc1' + acc_i         }
```

Listing 3: Loop summary example.

We pattern match accumulation operations within *ps* at line 9 and check whether a ϕ_m statement appears in the right-hand-side (rhs) precisely once. We also confirm that the rest of the expression, $\mathbb{E}_s$, is loop dependent using the loop-dependency set computed at line 8. If $\mathbb{E}_s$ depends on m which means that the loop violates the properties of an accumulation operation, we halt execution.

For handling nested summations, we consider every computed symbol *m1* with a summary *v*. We perform substitutions into the current summary *ps* and adjust the inner-sum's upper bound at lines 18-20. We also adjust previous summaries computed at line 22.

If m is an accumulator (*findSum* is True), we remove assignments to m from $\mathcal{S}^*$ so that no substitution is performed for already computed symbols (lines 25-26). Finally, to compute the full summary at end of the loop, we set the upper bound of the outermost summation to match the loop's upper bound (line 28).

In Listing 3, we show an example of a nested sum and the computed complete summaries for both *acc* and *acc1*.

Algorithm 2 *LpSm* procedure. It takes loop parameters and an expression $\mathbb{E}$ and returns an expression $\mathbb{E}'$. *M* stores symbols of $\mathbb{E}$, ordered based on $\mathcal{S}^*$. *V* stores summaries of symbols in *M*. *LD_i* stores expressions that are loop-dependent. *ApplyLDRules* updates *LD_i* using loop dependency rules.

1: **procedure** LpSm($i, n, \mathcal{S}^*, \mathbb{E}$)
2: $V \leftarrow \{\}$, $LD_i \leftarrow \{\}$
3: $\mathbb{E}' \leftarrow \mathbb{E}$
4: **for each** $m \in M$ **in their order of dependency**
5: $p_s \leftarrow m$
6: **for each** $stmt \in reverse(\mathcal{S}^*)$
7: $p_s \leftarrow$ ExtractSummary$(stmt, p_s)$
8: $LD_i \leftarrow$ ApplyLDRules$(stmt, LD_i)$ **using the rules L1-L4**
9: **if** $p_s \equiv$ "$\phi_m + \mathbb{E}_s$" $\wedge\ \mathbb{E}_s \in LD_i$
10: **if** ($\mathbb{E}_s$ **depends on** m)
11: **exit**
12: **else**
13: $i' \leftarrow$ NewIndex()
14: $ps \leftarrow m_0 + sum(\mathbb{E}_s[i/i'], i', i)$
15: $findSum \leftarrow True$
16: **for each** $(m1, v) \in V$
17: **if** $p_s \equiv$ "$m_0 + sum(\mathbb{E}_s, k, i)$" $\wedge\ m1 \in \mathbb{E}_s \wedge v \equiv$ "$v_0 + sum(\mathbb{E}_v, j, i)$"
18: $ps \leftarrow ps[m1/v[i/k]]$
19: **if** $p_s \equiv$ "$m_0 + sum(\mathbb{E}_s, k, i)$" $\wedge\ \phi_{m1} \in \mathbb{E}_s \wedge v \equiv$ "$v_0 + sum(\mathbb{E}_v, j, i)$"
20: $ps \leftarrow ps[\phi_{m1}/v[i/k-1]]$
21: **for each** $(m1, v) \in V$
22: **if** $p_s \equiv$ "$m_0 + sum(\mathbb{E}_s, k, i)$" $\wedge\ \phi_m \in v$
23: $V[m1] \leftarrow v[\phi_m/ps[i/i-1]]$
24: $V[m] \leftarrow ps$
25: **if** $findSum$
26: $A \leftarrow A \setminus assign(m, \_)$
27: **for each** $m \in M$
28: $v \leftarrow V[m][i/n]$
29: $\mathbb{E}' \leftarrow \mathbb{E}'[m/v]$
30: **return** $\mathbb{E}'$

Handling conditional branches. To account for the different branches of an *IfThenElse* construct, our summarization includes the condition in *IfThenElse*. Algorithm 3 describes the procedure to summarize the effects of *IfThenElse* construct. Similar to loops, we enumerate symbols in the order of dependency (line 4). We then handle statements in reverse order and generate a summary for each symbol. To combine summaries from both branches, we follow the *phi* instruction and take the condition into account (line 11).

```
for(uint i = 0; i < b; i++)  {
    if (D[i]) { a1 = a1 + A[i] * B[i]; }
    else      { a2 = a2 + A[i] * C[i]; }    }
```

Listing 4: If-else branch example.

In the example shown above, we generate the following summaries.

```
a1=a1_0+sum(index(A,j)*index(B,j)*Int(index(D,j)),j,b)
a2=a2_0+sum(index(A,k)*index(C,k)*(1-Int(index(D,k))),k,b)
```

Expressions in the if branch are multiplied by the Boolean flag, *D[i]*, while the ones in the else branch are multiplied by its complement.

4.2.3 Constraint Extraction. Our optimization model will consist of a set of constraints that are oracle-dependent or constraints over symbols that appear in oracle-dependent constraints. The first set of constraints corresponds to guard (require) statements that are flagged as oracle-dependent using the oracle dependency analysis rules *O1-O5*. The second set of constraints corresponds to guard statements that only contain symbols that appear in the set of oracle-dependent constraints.

Algorithm 3 *IfSm* procedure. It takes *IfThenElse* parameters and an expression $\mathbb{E}$ and returns an expression $\mathbb{E}'$. p_t and p_e represent the partial summary for the two branches. M_1 and M_2 stores the symbols of $\mathbb{E}$. Summaries of symbols in M_1 and M_2 are stored in *V*.

procedure $\textsc{IfSm}(\mathcal{E}_c, \mathcal{S}_1^*, \mathcal{S}_2^*, \Phi^*, \mathbb{E})$
  $V \leftarrow \{\}$
  $\mathbb{E}' \leftarrow \mathbb{E}$
  for each $m_1 \in M_1, m_2 \in M_2$ **in their order of dependency**
    $p_t \leftarrow m_1,\ p_e \leftarrow m_2$
    for each $stmt_t \in reverse(\mathcal{S}_1^*), stmt_e \in reverse(\mathcal{S}_2^*)$
      $p_t \leftarrow \textsc{ExtractSummary}(stmt_t, p_t)$
      $p_e \leftarrow \textsc{ExtractSummary}(stmt_e, p_e)$
    $V[m_1] \leftarrow p_t, V[m_2] \leftarrow p_e$
  for each $\phi_{id} \equiv phi(id_1, id_2) \in \Phi^*$
    $V[id] \leftarrow V[id_1] \times Int(\mathcal{E}_c) + V[id_2] \times (1 - Int(\mathcal{E}_c))$
  for each $m \in M$
    $\mathbb{E}' \leftarrow \mathbb{E}'[m/V[m]]$
  return $\mathbb{E}'$

Algorithm 4 *BuildModel* procedure. It takes in a list of constraints $ConstLst$ and returns a model *M* for the SMT solver.

1: **procedure** $\textsc{BuildModel}(ConstLst)$
2: $Cv, Sd, Re, Ub \leftarrow \textsc{ExtractVars}(ConstLst)$
3: $Cv, Sv, Re, Gt, \Delta \leftarrow \textsc{InitVar}(Cv, Sv, Re)$
4: $C0, C1 \leftarrow \textsc{InitConst}(Cv, Sv, Re, Gt, \Delta)$
5: $C \leftarrow [C0, C1]$
6: **for each** $const \in ConstLst$
7: $[C_{Re}, C_{Gt}] \leftarrow \textsc{ConvertZ3}(const, Re, Gt, Cv, Sv, Ub)$
8: $\textsc{append}(C, [C_{Re}, C_{Gt}])$
9: **return** $M(\Delta, Cv, Sv, Re, Gt, C, Ub)$

4.3 Model Generation

We build a model *M* from the extracted list of guard statements constraints. *M* has 7 parameters: Oracle prices values (*Re*), ground truth prices values (*Gt*), oracle deviation delta (Δ), the DeFi protocol's control variables (*Cv*), *e.g.*, margin ratio, state variables (*Sv*), loops upper bounds (*Ub*), and the set of constraints extracted *C*.

To generate *M* from the guard statements, we first extract and initialize variables (lines 2-3 in Algorithm 4). We also add two additional constraints *C0*, and *C1* to all models (lines 4-5). *C0* states that *Sv, Re, Gt* are greater than zero. *C1* states that *Re* deviates from *Gt* by at most Δ. For each guard statements, we generate two constraints, one evaluated with ground truth values and the other one with oracle values (lines 6-8).

4.4 Optimization

Ideally, we are interested in finding some oracle deviations Δ, or control variables *Cv*, such that the smart contracts "always behave correctly". In other words, for all inputs, given the deviated oracle price, the smart contracts should exhibit the same behavior as when given the ground truth price. For example, if we have a require statement: *require(a > b)*, the corresponding constraint is $a > b$, and assuming that one of the variables *a, b* or both of them are functions of oracle inputs. We need to prove the following.

$$a(Re) > b(Re) \Rightarrow a(Gt) > b(Gt) \quad (3)$$
$$a(Re) <= b(Re) \Rightarrow a(Gt) <= b(Gt) \quad (4)$$

Since we focus on the inputs when the transaction is not reverted, thus we only need to prove equation 3 (the require statement will revert the transaction if the lhs of equation 4 holds).

Algorithm 5 *SolvOpt* procedure. It takes in a model *M*, and returns the optimum parameters if found.

procedure $\textsc{SolvOpt}(M)$
  $ConsList \leftarrow \textsc{simplifyConstraints}(M)$
  while Stop condition not met
    $res \leftarrow \textsc{Solv}(ConsList)$
    $OptVar \leftarrow \textsc{Update}(res)$
  return $OptVar$

There are several optimization problems that can be derived from the constraints. For example, we can solve for the maximum oracle deviation the protocol can tolerate given some control parameters *Cv*, and *Ub* (equation 5). That is, we maximize the oracle deviation delta such that for all inputs satisfying $a > b$, given some predetermined control parameters. We can also give the model an expected delta and solve the optimization problem to find the optimum control parameters (equation 6). In Algorithm 4, we give the procedure *SolvOpt* that takes a model *M* and iteratively queries a solver to find optimum parameters, or it reaches a timeout.

$$\begin{aligned} &\max_{\Delta}\ \Delta \\ &\text{s.t. } \forall\ Sv > 0,\ \frac{|P_i - p_i|}{P_i} < \delta_i, \\ &a(Re, Sv, Cv) > b(Re, Sv, Cv) \Rightarrow \\ &a(Gt, Sv, Cv') > b(Gt, Sv, Cv') \end{aligned} \quad (5)$$

$$\begin{aligned} &\min_{Cv'}\ Cv' \\ &\text{s.t. } \forall\ Sv > 0,\ \frac{|P_i - p_i|}{P_i} < \delta_i, \\ &a(Re, Sv, Cv) > b(Re, Sv, Cv) \Rightarrow \\ &a(Gt, Sv, Cv') > b(Gt, Sv, Cv') \end{aligned} \quad (6)$$

5 EVALUATION

In this section, we evaluate the performance and effectiveness of OVer, and present the evaluation results. Specifically, we aim to answer the following research questions.

RQ 1: Are current control parameters of Defi protocols safe under large oracle deviations?
RQ 2: Can OVer efficiently analyze various Defi protocols that use oracles?
RQ 3: Can OVer assist developers to design safe Defi protocols that use oracles?

5.1 Implementation and Benchmarks

We implement OVer based on the *Slither* static analysis tool [40] with 1160 lines of code in *Python* for Solidity based smart contracts. To solve the optimization problems, we leverage the SMT solver *Z3* [16]. Note that the constructs of the programming language in Listing 1 that we used to present the main components of OVer design are commonly found in other programming languages. Thus, OVer implementation can also be extended to handle smart contracts written in other programming languages such as Vyper [45].

Table 2: Code summary module execution time.

Protocol	#requires	#loops	CompileTime (s)	TotalExecTime (s)	#vectorVars	#otherVars	Branch	Dependency	Oracle
Aave (borrow)	3	1	1.0558	1.0572	6	4	✓	✓	Chainlink
Aave (liquidation)	1	1	0.6313	0.6350	5	1	✓	✓	Chainlink
Compound	1	1	4.8403	4.8413	5	5	✓	✓	OpenPriceFeed
Euler	1	1	2.4056	2.4063	5	2	✓	✓	Uniswap
Solo	2	1	0.4704	0.4714	5	2	✓	✓	Chainlink
Warp	1	2	1.5149	1.5156	4	2	✓	✓	Uniswap
dForce	1	2	1.3724	1.3746	6	2	✓	✓	Chainlink
Morpho	1	2	8.5961	8.6002	7	1	✓	✓	Chainlink
TestAMM	1	0	0.1989	0.1992	0	4	X	✓	AMM-based
xToken	0	0	1.7244	1.7247	0	4	X	✓	multiple source
Beefy	0	0	0.6730	0.6750	0	4	X	✓	depends on vault

We evaluate OVer on 9 DeFi protocols: *Aave*, *Compound*, *Euler*, *Solo*, *Warp*, *dForce*, *Morpho*, *Beefy*, and *xToken*. Notably, this benchmark suite contains not only widely-used DeFi protocols according to DeFi industry database DefiLlama [17] but also that fell victim to oracle manipulation attacks. To the best of our knowledge, *Aave*, *Compound*, *Solo*, *Morpho*, and *Beefy* have not been victims to oracle manipulation attacks. The protocols that were victims to oracle manipulation attacks are *dForce*, *Warp*, *Euler*, and *xToken*. We cover a wide range of protocols, including different types of lending protocols, yield aggregators, margin trading, and liquidity manager. We excluded several DeFi protocols, *e.g.*, *Inverse Finance*, *CheeseBank*, *JustLend*, *Venus*, *Benqi*, and *Radiant*, that were forked from protocols in our benchmarks, *e.g.*, *Compound* and *Aave*. We also evaluate OVer on a fictional DeFi protocol [10] developed to demonstrate oracle manipulation, and we call it *TestAMM*. All experiments are run on an AWS EC2 m5.2xlarge instance machine with 8vCPU, 32 GB memory, and 8TB SSD storage.

5.2 Protocols' Response to Oracle Deviations

To motivate OVer and answer **RQ1**, we examine how oracle deviations impact the correctness of DeFi protocols. Specifically, we study historical oracle price deviations and the maximum tolerance of each protocol with their default control parameter settings.

To narrow the scope of our study, we focus on the oracle price of ETH, the native token of Ethereum network. We gather price updates for ETH/USD, USDT/ETH, USDC/ETH and DAI/ETH pair on *Chainlink* and ETH/USDT pair on *Uniswap*, where USDT, USDC and DAI are stable coins issued in Ethereum which stay closely one to one with US dollar. We select *Chainlink* and *Uniswap* because they are very widely used oracles among DeFi protocols [18], as also highlighted in the *oracle* column of Table 2.

Because empirically oracle deviations often occur when a digital asset is highly volatile, we study updates during the most volatile days of ETH for the two pairs between 06/2020 and 09/2022. We compute the deviation as the difference between two consecutive updates on Uniswap. The rationale is that in normal settings, the ground truth of an asset price is bounded by the values of two consecutive updates. On Chainlink, we look for deviations within 33 minutes or 155 blocks window.

Table 3 shows the top five deviations found. The first and the third columns give the block number when the deviation is observed on *Chainlink* and *Uniswap*, respectively. The second and fourth columns give the exact value of deviations.

Moreover, we study the maximum deviation allowed by each protocol. Since the lending protocols require over-collateralization to cover borrowed or leveraged positions, we define failure as when the borrowed value is more than the collateral value of the user. For *Aave*, *Morpho*, *Warp*, *dForce*, *Euler*, *xToken*, and *Beefy* we use the default control parameters of each protocol.

Table 4 shows the maximum tolerance of each protocol. Specifically, the first column gives the name of the protocol. The second column specifies the parameter used in the experiment. The last column presents the maximum oracle deviation found.

Table 3: Top deviations observed on Chainlink and Uniswap.

Chainlink	Deviation	Uniswap	Deviation
11631223	0.1390	10314022	0.4248
11631215	0.1293	10314022	0.3351
11631215	0.1260	10326501	0.2368
11631226	0.1159	10314022	0.2356
11631248	0.0994	10326310	0.1948

Table 4: Deviation limit given specific control variables.

Protocol	CV	delta
Compound	cf = 0.7	0.17
Aave	lth=0.85, ltv= 0.83	0.08
Solo	mp= 0.15, mr = 0.1	0
Morpho	ltv = 0.83	0.09
Warp	cr = 2/3	0.20
dForce	bf = 1, cf = 0.85	0.08
Euler	bf = 0.91, cf = 0.9	0.09
testAMM	cr = 0.7	0.42
xToken	fee = 0.02	0.02
Beefy	fee = 0	0

Answer to RQ1: We surprisingly found that the default control parameters of the investigated protocols are *not* enough to protect these protocols against history oracle deviation. Specifically, for protocols relying on *Chainlink* price feed, *e.g.*, *Aave* and *dForce*, with deviation limits of 0.08, will suffer from under-collateralization given the greatest deviation in Table 4. *Morpho* would encounter safety issues in certain cases. *Solo* is consistently at risk given the specific control parameters. *Compound*'s Open Price Feed module relies on *Chainlink* to update the price and verify it by comparing it with *Uniswap*'s average price. Thus, with a tolerance of 0.17, in some extreme cases, *Compound* would execute incorrectly. *Warp* and *Euler* use *Uniswap* as price oracle and *testAMM* relied on AMM-based oracles. Deviations on *Uniswap* are more significant, reaching a maximum value of 0.4248. While *testAMM* would not suffer from under-collateralization in most cases given the specific parameter, neither *Warp* nor *Euler* is safe given the oracle deviations.

This finding means that the oracle deviation caused these protocols, at least temporarily, to violate basic safety constraints such as over-collateralization. One consequence is, for example, that a malicious attacker could send timely transactions during the deviation to borrow or redeem assets with insufficient collaterals, extracting profits at the cost of the protocol investors.

In the case of *xToken* and *Beefy*, where the protocol does not mandate over-collateralization, any price deviation leads to an immediate loss. The protocols charge fees for most operations proportional to the transaction amount. *xToken* charges a maximum fee of 2%, while there is no deposit or withdraw fee in *Beefy* vaults. Consequently, if we employ the fee as a control parameter, the maximum oracle deviation the protocol can tolerate will correspond to the percentage of the fee. Furthermore, in some cases, fees can be exploited in an attack. An example is the fee adjustment from 0.5% to 0%, contributing to the *Yearn* attack in 2021 [2, 25].

Effect of Introducing Delay. Introducing a delay is a widely recommended approach to counteract oracle manipulation. An example of this strategy can be found in *MakerDao*'s OSM layer, which implements a one-hour delay for price updates. This approach naturally introduces a deviation to the reported oracle price. To evaluate this method, we conduct simulations using *Chainlink* data and calculate the deviation from the current timestamp when a one-hour delay is introduced. For instance, for the block with deviation of 0.1260, this strategy effectively reduces the deviation to 0.0779.

However, it is important to note that relying on a delayed price does not guarantee a consistently smaller deviation. It may introduce additional deviation due to the delay and therefore making the underlying protocols more vulnerable. For instance, during the period from block 11541949 to 11596096, we notice an increase in the maximum deviation from 0.0329 to 0.1525 after applying the delay method. This would make the deviation surpass the tolerable thresholds of five benchmark protocols, Aave, Morpho, dForce, Euler, and xToken shown in Table 4. This finding underscores the complexity of defending against oracle manipulation and shows that existing ad-hoc control mechanisms such as introducing delays are often insufficient or even detrimental to protect the DeFi protocols against the oracle deviation in the real-world.

5.3 Effectiveness of OVer

To answer **RQ2** and assess the performance of OVer, we run OVer to analyze the collected benchmarks. For the *Aave* protocol, we apply OVer to the safety constraint in both borrowing and liquidation scenarios, which are listed in rows one and two in Table 2, respectively. Notably, both Compound and Warp protocols share the same set of constraints for their borrow and liquidation operations. For *Solo*, we apply OVer on the safety constraint that verifies whether a user's position is adequately collateralized. The corresponding check is utilized in all operations, including liquidation, within the protocol. For *Euler*, we focus on the safety constraint responsible for checking liquidity in actions such as minting and withdrawal. As for *dForce*, *Morpho*, and *TestAMM*, we analyze the safety constraint of the borrow action. For *xToken* and *Beefy*, we focus on the constraint of the mint/deposit and burn/withdraw actions.

Table 2 presents the results of the experiment. The second and third columns present the number of "require" statements extracted and the occurrences of loops, respectively. We also include *Slither* compilation time in the fourth column and the total execution time in the fifth column. Moreover, we measure the number of vector variables and other variables (scalar) in the constraints (columns five and six). We also present the features of each benchmark, including branching and dependent statements.

Answer to RQ2: Our results highlight the capability of OVer. It successfully analyzed all the protocols and their safety constraints in less than 10 seconds. We manually validated all the generated symbolic expressions. For 8 out of the 11 cases, the contract code contains loops with dependencies or branch conditions. For 7 cases, the code contains more than one loop. Although the code structures are difficult for standard analysis techniques, our loop summary algorithm enables OVer to handle all of them successfully. Our loop summary algorithm is also fast, *i.e.*, the majority of the execution time is consumed by Slither to parse the code and generate AST.

5.4 Case Studies Analysis

To answer **RQ3** and show how OVer can help developers to design safe protocols, we present case studies of applying OVer on *Compound*, *Solo*, *dForce*, and *xToken*. For each case, we show how a user can use the symbolic expressions obtained by OVer to construct models to determine appropriate values of control parameters when facing different degrees of oracle deviations.

Timeout is set to be two minutes throughout the experiments.

Compound relies on the Open Price Feed module to access and retrieve price information critical to its operations. As discussed in Section 3, the protocol implements a one-side risk control mechanism, *i.e.*, uses a single control variable to govern its behavior. Specifically, the control variable is known as the collateral factor. Normally, cf is set to a value smaller than 1, ensuring that the user's collateral value exceeds the borrowed value. When cf is greater than 1, the protocol allows under-collateralization, a situation generally considered undesired for lending protocols. We set the cf to be 0.7 in the experiment, and consider three different oracle deviations. We run the search algorithm with a step size of 0.01 for $Ub{=}1$ and 0.05 for $Ub{=}2$. The results are shown in Table 5. The effective cf achieved, *i.e.*, cf', is given in the second column. The first column lists the parameter assignments, the third column counts the number of free variables in the constraint and the optimization time is shown in the last column. We time out when we set $bound{=}1, \delta{=}0.1$, and when we increase the bound Ub to 3. We observe that the result would be the same if we use the same step size. Furthermore, it is reasonable to argue that the same cf' would be optimal for $Ub{=}3$ as the search result should be independent of loop bounds.

Based on the results, if we expect an oracle deviation of 0.1 and set $cf = 0.7$ (equivalent to 30% safety margin), the actual margin will be around 14%, *i.e.*, $cf' = 0.86$. When there is no oracle deviation, we would achieve the exact margin specified in the protocol. This insight allows developers to understand how oracle deviations can impact the safety margin and offers guidance on parameter settings accordingly. Furthermore, developers can add the corresponding constraint on oracle inputs which would guarantee the correctness.

Table 5: Compound borrow with cf = 0.7 and ex = 1.

Variables	cf′	NumVars	Time (s)
$\delta = 0.1, bound = 1$	0.8600	5	4.5486
$\delta = 0.01, bound = 1$	0.7200	5	0.1194
$\delta = 0.001, bound = 1$	0.7100	5	0.0794
$\delta = 0.1, bound = 2$	NA	9	TO
$\delta = 0.01, bound = 2$	0.7150	9	0.3535
$\delta = 0.001, bound = 2$	0.7050	9	0.2316

Solo project[1] is a marginal trading protocol of *dXdY*, which uses *Chainlink* as price oracle. A desired property in *Solo* protocol is that for all operations, accounts remain in a collateralized position. Besides, for liquidation operation, the protocol may not want to execute unnecessary liquidation, thus verifies that the account being liquidated is indeed under-collateralized before proceeding with the action. Protocol developers employ two control parameters to safeguard operations: the margin ratio (*mr*) and the margin premium (*mp*). For liquidation to safely happen, the following constraint (simplified), extracted by OVer, must be met:

$$\frac{splyVal}{(1-mp)} < brwVal * (1+mp) * (1+mr)$$

where *splyVal* represents the total collateral and *brwVal* represents the total borrowed amount. These two variables are in the form of summation and are functions of oracle price input.

In the experiment, we set mr to 0.1 and mp to 0.15. The results are shown in Table 6, similar to Table 5, except columns two and three presents the *mp* and *mr* achieved. When the bound *Ub* is 1, we achieve the margin set in the protocol. However, as *Ub* is increased to 2, *mr* achieved, denoted as mr', also increases, resulting in a looser control effect. The experiment encountered a timeout when the *Ub* was further increased to 3.

Table 6: Solo liquidation with mp = 0.15 and mr = 0.1.

Variables	mp′	mr′	NumVars	Time (s)
$\delta = 0.1, bound = 1$	0.15	0.10	5	0.0280
$\delta = 0.01, bound = 1$	0.15	0.10	5	0.0281
$\delta = 0.001, bound = 1$	0.15	0.10	5	0.0281
$\delta = 0.1, bound = 2$	0.15	0.35	10	1.1197
$\delta = 0.01, bound = 2$	0.15	0.13	10	0.1454
$\delta = 0.001, bound = 2$	0.15	0.11	10	0.0888

dForce [22] is also a pool-based lending protocol and uses *Chainlink* as price oracle. While *dForce* also mandates over-collateralization, different from *Compound*, *dForce* designers enforce two-sided risk control, using 2 control variables, the collateral factor (*cf*) and the borrow factor (*bf*). OVer identifies the following safety constraint (simplified) in the smart contract to ensure collateralization:

$$cf * \sum_{c=0}^{cl}(cb[c] * pc[c]) > \sum_{b=0}^{bl}\frac{(bb[b] * pb[b])}{bf} \tag{7}$$

where *cb* and *bb* represent collateral and borrow balances, and *pc* and *pb* represent oracle prices. The constraint contains two summations, where bounds are represented by *cl* and *bl*, respectively.

While for most assets, the protocol did not use *bf* (*bf=1*), we set *cf*=0.5 and *bf*=0.7 for the experiment purposes. As there are 2 control variables to optimize, we fix one and search for the optimal value for the other. Table 7 shows the experiment results. Columns cf' and bf' show the *cf* and *bf* achieved. We observe that the results obtained are independent of the bounds for all cases *cl > 1, bl > 1.*

Table 7: dForce borrow with cf = 0.5 and bf = 0.7.

Variables	cf′	bf′	NumVars	Time (s)
$\delta = 0.1, cl = 1, bl = 0$	0.5	0.7	3	0.0264
$\delta = 0.01, cl = 1, bl = 0$	0.5	0.7	3	0.0265
$\delta = 0.001, cl = 1, bl = 0$	0.5	0.7	3	0.0263
$\delta = 0.1, cl = 1, bl = 1$	0.5	0.8560	6	3.8853
$\delta = 0.01, cl = 1, bl = 1$	0.5	0.7150	6	0.3996
$\delta = 0.001, cl = 1, bl = 1$	0.5	0.7020	6	0.1091
$\delta = 0.1, cl = 1, bl = 1$	0.6120	0.7	6	2.7787
$\delta = 0.01, cl = 1, bl = 1$	0.5110	0.7	6	0.3082
$\delta = 0.001, cl = 1, bl = 1$	0.5020	0.7	6	0.0892
$\delta = 0.1, cl = 2, bl = 1$	0.5	0.8560	9	7.1360
$\delta = 0.01, cl = 2, bl = 1$	0.5	0.7150	9	0.4700
$\delta = 0.001, cl = 2, bl = 1$	0.5	0.7020	9	0.0875
$\delta = 0.1, cl = 2, bl = 1$	0.6115	0.7	9	3.8853
$\delta = 0.01, cl = 2, bl = 1$	0.5110	0.7	9	0.3996
$\delta = 0.001, cl = 2, bl = 1$	0.5020	0.7	9	0.1091
$\delta = 0.1, cl = 2, bl = 2$	0.5	0.8560	12	9.0215
$\delta = 0.01, cl = 2, bl = 2$	0.5	0.7145	12	0.3842
$\delta = 0.001, cl = 2, bl = 2$	0.5	0.7015	12	0.1010
$\delta = 0.1, cl = 2, bl = 2$	0.6115	0.7	12	1.8996
$\delta = 0.01, cl = 2, bl = 2$	0.5105	0.7	12	0.6305
$\delta = 0.001, cl = 2, bl = 2$	0.5015	0.7	12	0.1676

xToken [24] serves as a liquidity manager protocol, and it was the victim of an oracle manipulation attack. Specifically, the attacker was able to arbitrage because the protocol utilizes different price sources. The common attack vector involves three steps: first, minting or depositing the token; second, inflating the price of the minted token; and finally, withdrawing or burning the token. Other protocols such as yield aggregators are susceptible to such attacks. To mitigate these attacks, we propose an interface that compares the price at the time of withdrawal to the price at the time of minting. The equation we suggest for this comparison is as follows:

$$\frac{|priceAtWithdraw - priceAtDeposit|}{priceAtDeposit} \leq \text{tol} \tag{8}$$

Many existing protocols rely on a fixed tolerance ratio, which is ineffective when a big volume of tokens is traded. Thus, it is crucial to parameterize the variable *tol* in order to take the amount of tokens traded into consideration. For example, we can parameterize *tol* as $\frac{profitAllowance}{tokenVolume}$, which restricts the profit of a single transaction.

We use OVer to examine mint, burn, deposit, and withdraw, and automatically extract the expression that approximates the price at withdraw and deposit, shown in Table 8. The price at deposit is approximated as the value transferred to the protocol and the token minted. Similarly, the withdraw price is represented by the value transferred to the user divided by token burned.

Table 8: Expressions extracted for xToken.

Protocol	mint	burn	NumVars	Time (s)
$xToken$	$\frac{etherContr}{mintAmt}$	$\frac{valToSend}{tokToRedeem}$	4	1.7247

Answer to RQ3: Our study shows that the analysis results of OVer can soundly capture the logic of the target safety constraints for various kinds of DeFi protocols. Given an oracle deviation ratio cap, a user can use the results of OVer to construct models to find optimal control parameters to guarantee the desired safety property.

6 RELATED WORK

Automatic Analysis. A significant body of research has been dedicated to the automatic auditing of smart contracts, utilizing classic methods such as fuzzing [12, 30, 39, 43, 49], symbolic executions [14, 34, 35, 38, 53], and static analysis [27, 31, 46, 48] to identify various vulnerabilities. Researchers have also built verification tools that use formal models to describe the intricate nature of these protocols and their interactions [7, 44, 47]. All the above work focus on eliminating or nullifying implementation errors in smart contracts. Furthermore, runtime validation techniques are adopted to enforce security constraints during the execution of smart contracts [23, 33, 42]. In contrast, we focus on the oracle deviation issue, which is the input aspect of the contract. We propose the first sound analysis tool to analyze oracle deviation in DeFi protocols.

Bartoletti, Massimo et. al. [6] propose a simulation-based approach for lending protocols, searching for optimal parameters to minimize non-repayable loans. In contrast, OVer works with existing `require` statements in the code, eliminating the need for explicit safety property specifications.

Oracle Design and Runtime Mechanisms. Extensive research has been conducted on DeFi protocols and the associated attacks, with recent emphasis on flash loan attacks, as highlighted in the work [11, 21, 41]. Additionally, the manipulation of oracles and price manipulation attacks have been extensively discussed. For instance, this work [36] demonstrates the vulnerability of lending protocols that employ TWAP oracles to undercollateralized loan attacks. Xue et. al. [52] suggest monitoring token changes in liquidity pools to detect anomalous transactions and proposes using front-running as a defense mechanism against such attacks. Wu et. al. [51] propose a framework for detecting oracle manipulation attacks through semantics recovery. An algorithmic model is designed to estimate the safety level of DEX-based oracles and calculate the cost associated with initiating price manipulation attacks [5].Wang et. al. develop a tool that detects price manipulation vulnerabilities by mutating states [50]. Several works focus on the design of robust oracles and proving the properties of price oracles [15]. While previous research has primarily concentrated on the design of robust oracles and the detection of price manipulation attacks, our work proposes promising analysis tools for smart contracts to help developers to mitigate oracle deviation caused by such attacks, operating under the assumption that oracles are unreliable.

Loop Summary. The loop summary component of our work is closely related to a previous work [37] which proposes a DSL containing map, zip, and fold operations and their variant to summarize *Solidity* loops. They use a type-directed search with an enumeration approach. However, after multiple experiments we are not able to use their tool, *Solis*, to implement the loop summary component of our work since it does not handle loops that contain `if-else` branches that require introducing Boolean flags in the summary. Furthermore, during our experiments, we faced some loops without if branches that require composition that cannot be handled using *Solis*. For example, the following loop requires composing the fold and zip operators on a single statement which according to Section 4.3 in [37] is not supported in *Solis*.

```
for (uint i = 0; i < len ; i ++) {
    total += arr1[i] * arr2[i];   }
```

Also, as presented in Section 4.3 in [37], *Solis* first generates a summary for a single statement and concatenates summaries through the sequence operator. Thus, it fails to handle dependent statements.

```
for (uint i = 0; i < len ; i ++)         {
    arr[i] * = 5; // S1
    total += arr[i]; //S2 depends on S1    }
```

In DeFi protocols, most loops perform fold operations and include complex mathematical expressions. Therefore, we develop a new loop summarization algorithm that is tailored to DeFi smart contracts to address the above issues.

7 THREATS TO VALIDITY

One threat to the validity of our results is that OVer might not be able to analyze the source code of DeFi protocols beyond our benchmark set. To mitigate this threat, we curated a diverse collection of benchmark protocols that manage digital assets worth billions of dollars. Another potential threat is that we focus on the five most volatile days in history to estimate the upper limit of oracle deviations. However, even if we overlooked certain data points, it does not undermine our surprising finding that the current control mechanisms in many deployed DeFi protocols are inadequate for protecting the protocols against oracle deviations.

8 CONCLUSION

The integrity of decentralized finance protocols is frequently contingent on the precision of crucial oracle values, such as the prices of digital assets. In response to this, we introduced OVer, the first sound analysis tool that aids developers in constructing formal models directly from contract source code. Our findings demonstrate that OVer possesses the capability to analyze a broad spectrum of prevalent DeFi protocols. Intriguingly, with the assistance of OVer, we discovered that many existing DeFi protocols' control mechanisms, even with default parameters, fall short in safeguarding the protocols against historical oracle deviations. This revelation underscores the indispensable role of tools like OVer and advocates for more methodical strategies in the design of DeFi protocols.

9 DATA AVAILABILITY

Our artifact includes the implementation of OVer, source code for benchmark protocols and the experiment data. It is publicly accessible on Zenodo [19]. Furthermore, an extended version of the paper, including supplementary experiment results, can be found on arxiv [20].

ACKNOWLEDGEMENT

We thank anonymous reviewers for their insightful comments on the early version of the paper. This work was supported by Mitacs through the Mitacs Accelerate program. Sidi Mohamed Beillahi is supported by NSERC Postdoctoral Fellowship.

REFERENCES

[1] 2021. Solo protocol. https://github.com/dydxprotocol/solo/releases/tag/v0.41.0.

[2] 2023. Yearn Attack Disclosure. https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md.

[3] Aave. 2023. Aave V2. https://github.com/aave/protocol-v2/tree/master.

[4] John Adler, Ryan Berryhill, Andreas Veneris, Zissis Poulos, Neil Veira, and Anastasia Kastania. 2018. Astraea: A decentralized blockchain oracle. In *2018 IEEE international conference on internet of things (iThings) and IEEE green computing and communications (GreenCom) and IEEE cyber, physical and social computing (CPSCom) and IEEE smart data (SmartData)*. IEEE, 1145–1152.

[5] Ayana T Aspembitova and Michael A Bentley. 2022. Oracles in Decentralized Finance: Attack Costs, Profits and Mitigation Measures. *Entropy* 25, 1 (2022), 60.

[6] Massimo Bartoletti, James Chiang, Tommi Junttila, Alberto Lluch Lafuente, Massimiliano Mirelli, and Andrea Vandin. 2022. Formal analysis of lending pools in decentralized finance. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 335–355.

[7] Thomas Bernardi, Nurit Dor, Anastasia Fedotov, Shelly Grossman, Neil Immerman, Daniel Jackson, Alexander Nutz, Lior Oppenheim, Or Pistiner, Noam Rinetzky, et al. 2020. WIP: Finding bugs automatically in smart contracts with parameterized invariants. *Retrieved July* 14 (2020), 2020.

[8] Vitalik Buterin. 2014. Ethereum: A next-generation smart contract and decentralized application platform. https://ethereum.org/en/whitepaper/.

[9] Yuxi Cai, Nafis Irtija, Eirini Eleni Tsiropoulou, and Andreas Veneris. 2022. Truthful decentralized blockchain oracles. *International Journal of Network Management* 32, 2 (2022), e2179.

[10] calvwang9. 2022. Oracle Manipulation. https://github.com/calvwang9/oracle-manipulation.

[11] Zhiyang Chen, Sidi Mohamed Beillahi, and Fan Long. 2022. FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation. *arXiv preprint arXiv:2206.10708* (2022).

[12] 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.

[13] Compound Finance. 2020. Compound V2. https://github.com/compound-finance/compound-protocol/releases/tag/v2.8.1.

[14] Consensys. 2023. Mythril: a security analysis tool for EVM bytecode. https://github.com/Consensys/mythril.

[15] Kinnari Dave, Vilhelm Sjöberg, and Xinyuan Sun. 2021. Towards Verified Price Oracles for Decentralized Exchange Protocols. In *3rd International Workshop on Formal Methods for Blockchains (FMBC 2021) (Open Access Series in Informatics (OASIcs), Vol. 95)*, Bruno Bernardo and Diego Marmsoler (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1:1–1:14. https://doi.org/10.4230/OASIcs.FMBC.2021.1

[16] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340.

[17] DeFiLlama. 2023. DeFiLlama - DeFi Dashboard. https://defillama.com/.

[18] DeFiLlama. 2023. DeFiLlama - Oracles Dashboard. https://defillama.com/oracles.

[19] Xun Deng, Sidi Mohamed Beillahi, Cyrus Minwalla, Han Du, Andreas Veneris, and Fan Long. 2024. *Artifact for OVer: Safeguarding DeFi Smart Contracts against Oracle Deviations*. https://doi.org/10.5281/zenodo.10436720

[20] Xun Deng, Sidi Mohamed Beillahi, Cyrus Minwalla, Han Du, Andreas Veneris, and Fan Long. 2024. Safeguarding DeFi Smart Contracts against Oracle Deviations. arXiv:2401.06044 [cs.SE]

[21] Xun Deng, Zihan Zhao, Sidi Mohamed Beillahi, Han Du, Cyrus Minwalla, Keerthi Nelaturu, Andreas Veneris, and Fan Long. 2023. A Robust Front-Running Methodology for Malicious Flash-Loan DeFi Attacks. In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*. IEEE, 38–47.

[22] dforce Network. 2021. Lending Contracts. https://github.com/dforce-network/LendingContractsV2/tree/master/contracts.

[23] Joshua Ellul and Gordon J Pace. 2018. Runtime verification of ethereum smart contracts. In *2018 14th European Dependable Computing Conference (EDCC)*. IEEE, 158–163.

[24] Etherscanners. 2020. xToken Victim Contract. https://etherscan.io/address/0x04bef870de607519c91d16a23434ad5745f62a63#code.

[25] Etherscanners. 2023. Yearn Attack. https://etherscan.io/tx/0xf6022012b73770e7e2177129e648980a82aab555f9ac88b8a9cda3ec44b30779.

[26] Euler. 2023. Euler Smart Contracts. https://github.com/euler-xyz/euler-contracts.

[27] 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.

[28] Chainlink Foundation. 2023. Chainlink API. https://docs.chain.link/any-api/api-reference/.

[29] Ethereum Foundation. 2023. The Solidity Contract-Oriented Programming Language. https://github.com/ethereum/solidity

[30] Bo Jiang, Ye Liu, and Wing Kwong Chan. 2018. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 259–269.

[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: analyzing safety of smart contracts.. In *Ndss*. 1–12.

[32] Uniswap Labs. 2023. Uniswap Protocol. https://uniswap.org/.

[33] Ao Li, Jemin Andrew Choi, and Fan Long. 2020. Securing smart contract with runtime validation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 438–453.

[34] 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.

[35] 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.

[36] Torgin Mackinga, Tejaswi Nadahalli, and Roger Wattenhofer. 2022. TWAP Oracle Attacks: Easier Done than Said?. In *2022 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 1–8. https://doi.org/10.1109/ICBC54727.2022.9805499

[37] Benjamin Mariano, Yanju Chen, Yu Feng, Shuvendu K. Lahiri, and Isil Dillig. 2021. Demystifying Loops in Smart Contracts *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 262–274. https://doi.org/10.1145/3324884.3416626

[38] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[39] 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.

[40] Trail of Bits. 2023. Slither: Static Analyzer for Solidity. https://github.com/crytic/slither.

[41] Kaihua Qin, Liyi Zhou, Benjamin Livshits, and Arthur Gervais. 2021. Attacking the defi ecosystem with flash loans for fun and profit. In *International conference on financial cryptography and data security*. Springer, 3–32.

[42] Michael Rodler, Wenting Li, Ghassan O Karame, and Lucas Davi. 2018. Sereum: Protecting existing smart contracts against re-entrancy attacks. *arXiv preprint arXiv:1812.05934* (2018).

[43] Chaofan Shou, Shangyin Tan, and Koushik Sen. 2023. ItyFuzz: Snapshot-Based Fuzzer for Smart Contract. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 322–333.

[44] Tianyu Sun and Wensheng Yu. 2020. A formal verification framework for security issues of blockchain smart contracts. *Electronics* 9, 2 (2020), 255.

[45] Vyper Team. 2023. Vyper. https://vyper.readthedocs.io/en/stable/

[46] 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.

[47] Palina Tolmach, Yi Li, Shang-Wei Lin, and Yang Liu. 2021. Formal analysis of composable DeFi protocols. In *Financial Cryptography and Data Security. FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Virtual Event, March 5, 2021, Revised Selected Papers 25*. Springer, 149–161.

[48] 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.

[49] Haijun Wang, Ye Liu, Yi Li, Shang-Wei Lin, Cyrille Artho, Lei Ma, and Yang Liu. 2020. Oracle-supported dynamic exploit generation for smart contracts. *IEEE Transactions on Dependable and Secure Computing* 19, 3 (2020), 1795–1809.

[50] Shih-Hung Wang, Chia-Chien Wu, Yu-Chuan Liang, Li-Hsun Hsieh, and Hsu-Chun Hsiao. 2021. ProMutator: Detecting vulnerable price oracles in DeFi by mutated transactions. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 380–385.

[51] 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:2104.15068 [cs.CR]

[52] Yue Xue, Jialu Fu, Shen Su, Zakirul Alam Bhuiyan, Jing Qiu, Hui Lu, Ning Hu, and Zhihong Tian. 2022. Preventing Price Manipulation Attack by Front-Running. In *International Conference on Artificial Intelligence and Security*. Springer, 309–322.

[53] Peilin Zheng, Zibin Zheng, and Xiapu Luo. 2022. Park: Accelerating smart contract vulnerability detection via parallel-fork symbolic execution. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 740–751.