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

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 multifaceted 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.
function borrowAllowed(address cToken, address brw, uint brwAmt) returns (uint) {
  ... // Return true if borrowing is allowed.
  uint surplus = hypotheticalLiquid(brw, cToken, 0, brwAmt);
  require((surplus > 0), "INSUFFICIENT_LIQUIDITY");
  ... // Perform additional checks...
  return uint(Error.NO_ERROR); }

function hypotheticalLiquid(address acct, CToken cToken, uint redTok, uint brwAmt) returns (uint) {
  AccountLiquidLocalVars memory v;
  // Iterate over each asset in the acct
  CToken[] memory assets = accountAssets[acct];
  for (uint i = 0; i < assets.length; i++) {
    CToken asset = assets[i];
    (... // Fetch asset price...
    v.oraclePrice = oracle.getUnderlyingPrice(asset);
    v.collFact = markets[address(asset)].collFact;
    v.tokensToDenom = v.collFact*v.exchRt*v.oraclePrice;
    v.sumColl = v.sumColl + v.tokensToDenom * v.cTokenBal;
    v.tokensToDenom = v.collFact * v.exchRt * v.oraclePrice;
    v.sumBrwEfct = v.sumBrwEfct + v.oraclePrice * v.brwBal;
    if (asset == cToken) {
      v.sumBrwEfct = v.sumBrwEfct + v.oraclePrice * v.brwBal;
      if (surplus > 0) {
        v.sumBrwEfct = v.sumBrwEfct + v.oraclePrice * v.brwBal;
      }
    } else {
      v.sumBrwEfct = v.sumBrwEfct + v.oraclePrice * v.brwBal;
    }
    v.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 code 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:

\[
\sum_{assets} \left( \text{collFact} \times \text{exchRt} \times p_i \times \text{cTokenBal}_i \right)
\]

(1)

where \( \text{exchRt} \) is the exchange rate of the collateral asset \( a \), \( p_i \) is the price of the asset \( a \) fetched from an external oracle contract \( v.\text{oraclePrice} \) at line 20, \( c\text{TokenBal}_i \) is the balance of the asset \( a \), and \( \text{collFact} \) is a control variable smaller than one to determine the enforced over-collateralization ratio for the asset. The loop also computes \( v.\text{sumBrwEfct} \) as follows:

\[
\sum_{assets} \left( \text{brwBal}_i \times p_i + c_a \times (p_{brw} \times \text{brwBal}_i + \text{collFact} \times \text{exchRt} \times p_i \times \text{redTok}) \right)
\]

(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, \( \text{brwBal}_i \) is the already borrowed balance of the asset \( a \), \( \text{brwBal}_i \) is the amount of the asset \( brw \) a user wants to borrow, and \( \text{redTok} \) is the amount of asset \( r \) a user wants to redeem. \( c_a \) is a binary representation of the condition at line 22, where \( c_a = 1 \) when the asset \( a \) is \( c\text{Token} \) 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 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 \( v.\text{sumBrwEfct} \) from \( v.\text{sumColl} \). The expressions extracted by OVer for \( v.\text{sumColl} \) and \( v.\text{sumBrwEfct} \) 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
expression are either loaded contract states (e.g., v.brwBal) or the return values of external function calls (e.g., getUnderlyingPrice).

Loop summarization. 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:

\[
\sum_{\text{assets}}(\text{collFact}_a \times \text{exchRt}_a \times p_a \times c\text{TokenBal}_a) - (\sum_{\text{assets}}(\text{brwBal}_a \times p_a + c_a \times (p_{brw} \times \text{brwAmt} + \text{collFact}_a \times \text{exchRt}_a \times p_i))) > 0
\]

Figure 2: Compound analysis summary.

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 \(i\), the loop condition \(P\), the loop control \(b\), and the loop body \(\text{block}\). The function call \(\text{require}\) enforces smart contract constraints and its logic, and is crucial in our analysis.

Function Calls. Function calls are represented by the statement \(\text{call}(f, E^*, id^*)\), where \(f\) identifies the function, \(E^*\) 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 procedure, we consider those as free variables. Table 1 shows examples of statements from the Compound protocol.

upper-bound \(n\) (for simplicity we omitted lower-bound), and a sequence of statements representing the loop body. A phi instruction, denoted as \(\phi_{id}\) 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.

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 mechanism1. 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.

1In 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.
4.2 Code Summary Module

The 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.

#### Listing 2: Code summary DSL.

The two main components of our DSL are: the expressions and constraints for boolean constraint expressions. The expression for a boolean expression is either an accumulation value, a value or an arithmetic expression (or). The type of an expression 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 \( \text{index}(V_1, V_2) \) to represent accessing an element from the array or map \( V_1 \) with the key \( V_2 \). The type of \( V_1 \) must be either an array or map and the type of \( V_2 \) must be the same as the key’s type of \( V_1 \). This definition of the index operator allows nested indexing. The member-access operator \( V_1(V_2) \) is used to represent accessing the field \( V_1 \) of the struct variable \( V_2 \).

**Accumulation value.** To represent a loop’s summary in our DSL, we use the accumulation operator \( \text{sum}(\mathbb{E}, i, ub) \), 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 \( \text{ret}(V, i) \) to indicate that \( V \) is the return value of a pure function which reads global states. When \( V \) is a loop-dependent, \( i \) represents the loop iterator, otherwise \( i \) is null. For example, at line 18 in Figure 1, \( \text{v.oraclePrice} \) gets the value from the function \( \text{oracle.getUnderlyingPrice} \) and the function is loop-dependent because of the argument \( \text{asset} \). The generated summary is \( \text{ret} \langle \text{oraclePrice}(v), i \rangle \).

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

**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.

\[
S := \text{load}(i, s), s \in OD \quad \Rightarrow \quad S := \text{assign}(id, E), E \in OD
\]

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

\[
E = E_1 \text{ op } E_2, E_1 \in OD \lor E_2 \in OD \quad \Rightarrow \quad E \in OD \lor \{id, S\}
\]

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

\[
S := \text{call}(f, E, id), f := \text{func}(f, S^*), E \in OD \lor S^* \in OD \quad \Rightarrow \quad OD = OD \lor \{id, S\}
\]

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

\[
S := \text{require}(E), E \in OD \quad \Rightarrow \quad OD = OD \lor \{S\}
\]

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

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

\[
E := E_1 \text{ op } E_2, E_1 \in LD_i \lor E_2 \in LD_i \quad \Rightarrow \quad LD_i = LD_i \lor \{E\}
\]

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

\[
E := E_1 \text{ op } E_2, E_1 \in LD_i \lor E_2 \in LD_i \quad \Rightarrow \quad LD_i = LD_i \lor \{E\}
\]

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

\[
S := \text{assign}(id, E), E \in LD_i \quad \Rightarrow \quad LD_i = LD_i \lor \{id\}
\]

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

<table>
<thead>
<tr>
<th>Solidity Code</th>
<th>Simplified Solidity</th>
</tr>
</thead>
<tbody>
<tr>
<td>require(surplus &gt; 0, &quot;INSUFFICIENT_LIQUIDITY&quot;)</td>
<td>require(surplus &gt; 0)</td>
</tr>
</tbody>
</table>
pc(f) := require E or pc(f) := return E, S(f) = L, E := ConoDSL(E)\\n\hspace{1cm} S = [f \mapsto E]\\n\hspace{1cm} pc(f) := assign(id,E), S(f) = [E], E' := [id|ConoDSL(E)]\\n\hspace{1cm} S = [f \mapsto E']\\n\hspace{1cm} pc(f) := call(f', E, id), S(f) = B, E' = B[id|S(f') (E)]\\n\hspace{1cm} S = [f \mapsto E']\\n\hspace{1cm} pc(f) := for(i,n) (S'), E := LpSm(i,n,S', S(f))\\n\hspace{1cm} S = [f \mapsto E']\\n\hspace{1cm} pc(f) := if(\phi_0) then\{S_1\} else\{S_2\}, E := LpSm(\phi_0, S_1, S_2, \Phi, S(f))\\n\hspace{1cm} S = [f \mapsto E']

Figure 4: Function summary extraction rules.

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

\[ S := call(f,E, id), E \in LD_i \]

\[ LD_i = LD_i \cup \{id\} \]

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 S and an expression E. It applies the effect of S on E and returns the updated expression E'. We use S̄ 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 Ē of a function f to the initially empty summary S̄(f).

For an assignment, id = E, we convert E to its DSL using the procedure ConoDSL. We substitute all occurrences of id in S̄(f) = B̄ with its computed DSL value, denoted as E[id|ConoDSL(E)̄].

For a function call, call(f,E, id), we generate a summary of f(E), denoted as S̄(f(E)), and replace symbolic id with S̄(f(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 Ē, 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.

```
1 \text{acc: acc} = \text{acc}_0 + \text{sum(index}(A, i, j), b) + \text{acc'}, \text{acc}_0 = \text{acc}_0 + \text{sum(index}(A, i, j), k, b)
2 \text{for } (i = 0; i < b; i++) 
3 \text{acc'} = \phi_{\text{acc}_0, \text{acc}_{i-1}}
4 \text{acc' = acc'} + A[i]
5 \text{acc'' = acc' + acc_i}
```

Listing 3: Loop summary example.

We pattern match accumulation operations within ps at line 9 and check whether a \( \phi_m \) statement appears in the right-hand-side (rhs) precisely once. We also confirm that the rest of the expression, \( E_\text{s} \), is loop dependent using the loop-dependency set computed at line 8. If \( E_\text{s} \) 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 \( m \) 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 (\( \text{findSum} \) is True), we remove assignments to \( m \) from \( E_\text{s} \) 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 Ē and returns an expression S̄, M stores symbols of Ē, ordered based on S̄. V stores summaries of symbols in M. LD_i stores expressions that are loop-dependent. ApplyLDRules updates LD_i using loop dependency rules.

```python
1: \text{procedure LpSm}(n, S', E)\\n2: \hspace{1cm} V \leftarrow \{\}, \text{LD}_i \leftarrow \{\}
3: \hspace{1cm} E' \leftarrow E\\n4: \text{for each } m \in M \text{ in their order of dependency}\\n5: \hspace{1cm} p_S \leftarrow m\\n6: \hspace{1cm} \text{for each } \text{stmt} \in \text{reverse}(S')\\n7: \hspace{1cm} p_S \leftarrow \text{ApplyLDRules}(\text{stmt}, p_S)\\n8: \hspace{1cm} \text{LD}_i \leftarrow \text{ApplyLDRules}(\text{stmt}, \text{LD}_i)\\n9: \hspace{1cm} p_S \leftarrow p_S^{\phi_m + E_\text{s}} \wedge E_\text{s} \in \text{LD}_i\\n10: \hspace{1cm} \text{if } (E_\text{s} \text{ depends on } m)\\n11: \hspace{1cm} \text{else}\\n12: \hspace{1cm} \text{exit}\\n13: \hspace{1cm} i' \leftarrow \text{NewIndex()}\\n14: \hspace{1cm} p_S \leftarrow m + \text{sum}(E_\text{s}, i', i, i')\\n15: \hspace{1cm} \text{if } \text{findSum} \leftarrow \text{True}\\n16: \hspace{1cm} \text{for each } (m, v) \in V\\n17: \hspace{1cm} \hspace{1cm} p_S \leftarrow m + \text{sum}(E_\text{s}, k, i') \
18: \hspace{1cm} \hspace{1cm} \text{and } m_1 \in E_\text{s} \
19: \hspace{1cm} \hspace{1cm} \text{and } v \equiv \text{"m_0 + sum(E_0, j, i)"}\\n20: \hspace{1cm} \hspace{1cm} p_S \leftarrow p_S[m_1|a[i/k]]\\n21: \hspace{1cm} \text{and } p_S \leftarrow p_S^{\phi_m + E_\text{s}} \wedge E_\text{s} \in \text{LD}_i\\n22: \hspace{1cm} \text{if } (m, v) \in \text{V}\\n23: \hspace{1cm} \hspace{1cm} p_S \leftarrow p_S^{\phi_m/p_S[i/i'-1]}\\n24: \hspace{1cm} \text{V}[m] \leftarrow p_S\\n25: \hspace{1cm} \text{if } \text{findSum}\\n26: \hspace{1cm} \text{A} \leftarrow A \setminus \{\text{assign}(m, \_}\)\\n27: \hspace{1cm} \text{for each } m \in M\\n28: \hspace{1cm} \hspace{1cm} v \leftarrow V[m]/i/n\\n29: \hspace{1cm} \hspace{1cm} E' \leftarrow E'[m/v]\\n30: \hspace{1cm} \text{return } 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 \( \phi_i \) instruction and take the condition into account (line 11).

```
1: \text{for } \text{uint } i = 0; i < b; i++ 
2: \hspace{1cm} \text{if } (D[i]) \{ a_1 = a_1 + \text{A[i]} \times B[i]; \}
3: \hspace{1cm} \text{else} \{ a_2 = a_2 + \text{A[i]} \times C[i]; \}
```

Listing 4: If-else branch example.

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

```
a_1 = a_1 + \text{sum(index}(A, j), j) \times \text{Int(index}(D[i]));
1.b = a_2 = a_2 + \text{sum(index}(A, k), k) \times \text{Int(index}(D[i], k));
```

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 InSm procedure. It takes IfThenElse parameters and an expression E and returns an expression E’. p∗ and p# represent the partial summary for the two branches. M1 and M2 stores the symbols of E. Summaries of symbols in M1 and M2 are stored in V. To generate M from the guard statements, we first extract and convert Z3 parameters: Oracle prices values (Cv), 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.

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 Cφ and C1 to all models (lines 4-5). Cφ 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.

1. a(Re) > b(Re) ⇒ a(Gt) > b(Gt)
2. a(Re) < b(Re) ⇒ a(Gt) < b(Gt)

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 4 BuildModel (ConstList) and returns a model M for the SMT solver.

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

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.

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].
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 2: Code summary module execution time.

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

Table 3: Top deviations observed on Chainlink and Uniswap.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Deviation</th>
<th>Uniswap</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aave</td>
<td>0.1390</td>
<td>0.1022</td>
<td>0.4248</td>
</tr>
<tr>
<td>Compound</td>
<td>0.1293</td>
<td>0.1022</td>
<td>0.3351</td>
</tr>
<tr>
<td>Solo</td>
<td>0.1260</td>
<td>0.1022</td>
<td>0.2368</td>
</tr>
<tr>
<td>Warp</td>
<td>0.1159</td>
<td>0.1022</td>
<td>0.2356</td>
</tr>
<tr>
<td>dForce</td>
<td>0.0994</td>
<td>0.1022</td>
<td>0.1948</td>
</tr>
</tbody>
</table>

Table 4: Deviation limit given specific control variables.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>CV</th>
<th>delta</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compound</td>
<td>cf = 0.7</td>
<td>0.17</td>
</tr>
<tr>
<td>Aave</td>
<td>ltv=0.85, ltv=0.83</td>
<td>0.08</td>
</tr>
<tr>
<td>Solo</td>
<td>mp=0.15, mr=0.1</td>
<td>0</td>
</tr>
<tr>
<td>Morpho</td>
<td>ltv=0.83</td>
<td>0.09</td>
</tr>
<tr>
<td>Warp</td>
<td>cr = 2/3</td>
<td>0.20</td>
</tr>
<tr>
<td>dForce</td>
<td>bf = 1, cf = 0.85</td>
<td>0.08</td>
</tr>
<tr>
<td>Euler</td>
<td>bf = 0.91, cf = 0.9</td>
<td>0.09</td>
</tr>
<tr>
<td>testAMM</td>
<td>cr = 0.7</td>
<td>0.42</td>
</tr>
<tr>
<td>xToken</td>
<td>fee = 0.02</td>
<td>0.02</td>
</tr>
<tr>
<td>Beefy</td>
<td>fee = 0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

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 vector variables in the constraint and the optimization time is shown in the last column. We time out when we set bound=1, δ=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>
<thead>
<tr>
<th>Variables</th>
<th>$cf^\gamma$</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta = 0.1, \text{bound} = 1$</td>
<td>0.8600</td>
<td>5</td>
<td>4.5486</td>
</tr>
<tr>
<td>$\delta = 0.01, \text{bound} = 1$</td>
<td>0.7200</td>
<td>5</td>
<td>0.1194</td>
</tr>
<tr>
<td>$\delta = 0.001, \text{bound} = 1$</td>
<td>0.7100</td>
<td>5</td>
<td>0.0794</td>
</tr>
<tr>
<td>$\delta = 0.1, \text{bound} = 2$</td>
<td>NA</td>
<td>9</td>
<td>TO</td>
</tr>
<tr>
<td>$\delta = 0.01, \text{bound} = 2$</td>
<td>0.7150</td>
<td>5</td>
<td>0.3535</td>
</tr>
<tr>
<td>$\delta = 0.001, \text{bound} = 2$</td>
<td>0.7050</td>
<td>9</td>
<td>0.2316</td>
</tr>
</tbody>
</table>

Solo project[1] is a marginal trading protocol of dX/dY, 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:

$$\text{splyVal} \leq \text{brwVal} + (1 + mp) \times (1 + mr)$$

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>
<thead>
<tr>
<th>Variables</th>
<th>$mp^\gamma$</th>
<th>$mr^\gamma$</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta = 0.1, \text{bound} = 1$</td>
<td>0.15</td>
<td>0.10</td>
<td>5</td>
<td>0.0280</td>
</tr>
<tr>
<td>$\delta = 0.01, \text{bound} = 1$</td>
<td>0.15</td>
<td>0.10</td>
<td>5</td>
<td>0.0281</td>
</tr>
<tr>
<td>$\delta = 0.001, \text{bound} = 1$</td>
<td>0.15</td>
<td>0.10</td>
<td>5</td>
<td>0.0281</td>
</tr>
<tr>
<td>$\delta = 0.1, \text{bound} = 2$</td>
<td>0.15</td>
<td>0.35</td>
<td>10</td>
<td>1.1197</td>
</tr>
<tr>
<td>$\delta = 0.01, \text{bound} = 2$</td>
<td>0.15</td>
<td>0.13</td>
<td>10</td>
<td>0.1454</td>
</tr>
<tr>
<td>$\delta = 0.001, \text{bound} = 2$</td>
<td>0.15</td>
<td>0.11</td>
<td>10</td>
<td>0.0888</td>
</tr>
</tbody>
</table>

$dForce$ [2] is also a pool-based lending protocol and uses Chainlink as price oracle. While $dForce$ also mandates over-collateralization, different from Compound, $dForce$ enforces 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:

$$cf \times \sum_{c=0}^{cl} (cb[c] \times pc[c]) > \sum_{b=0}^{bl} (bb[b] \times pb[b]) / bf$$

(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 $bf$ ($bf=0$), 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>
<thead>
<tr>
<th>Variables</th>
<th>$cf'$</th>
<th>$bf'$</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta = 0.1, cl = 1, bl = 0$</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0264</td>
</tr>
<tr>
<td>$\delta = 0.01, cl = 1, bl = 0$</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0265</td>
</tr>
<tr>
<td>$\delta = 0.001, cl = 1, bl = 0$</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0263</td>
</tr>
<tr>
<td>$\delta = 0.1, cl = 1, bl = 1$</td>
<td>0.5</td>
<td>0.8560</td>
<td>6</td>
<td>3.8853</td>
</tr>
<tr>
<td>$\delta = 0.01, cl = 1, bl = 1$</td>
<td>0.5</td>
<td>0.7150</td>
<td>6</td>
<td>0.3996</td>
</tr>
<tr>
<td>$\delta = 0.001, cl = 1, bl = 1$</td>
<td>0.5</td>
<td>0.7020</td>
<td>6</td>
<td>0.1091</td>
</tr>
<tr>
<td>$\delta = 0.1, cl = 2, bl = 1$</td>
<td>0.6120</td>
<td>0.7</td>
<td>6</td>
<td>2.7787</td>
</tr>
<tr>
<td>$\delta = 0.01, cl = 2, bl = 1$</td>
<td>0.5110</td>
<td>0.7</td>
<td>6</td>
<td>0.3802</td>
</tr>
<tr>
<td>$\delta = 0.001, cl = 2, bl = 1$</td>
<td>0.5020</td>
<td>0.7</td>
<td>6</td>
<td>0.0987</td>
</tr>
<tr>
<td>$\delta = 0.1, cl = 2, bl = 2$</td>
<td>0.6115</td>
<td>0.7</td>
<td>9</td>
<td>3.8853</td>
</tr>
<tr>
<td>$\delta = 0.01, cl = 2, bl = 2$</td>
<td>0.5110</td>
<td>0.7</td>
<td>9</td>
<td>0.3996</td>
</tr>
<tr>
<td>$\delta = 0.001, cl = 2, bl = 2$</td>
<td>0.5020</td>
<td>0.7</td>
<td>9</td>
<td>0.1091</td>
</tr>
<tr>
<td>$\delta = 0.1, cl = 2, bl = 2$</td>
<td>0.6115</td>
<td>0.7</td>
<td>12</td>
<td>1.8996</td>
</tr>
<tr>
<td>$\delta = 0.01, cl = 2, bl = 2$</td>
<td>0.5105</td>
<td>0.7</td>
<td>12</td>
<td>0.6305</td>
</tr>
<tr>
<td>$\delta = 0.001, cl = 2, bl = 2$</td>
<td>0.5015</td>
<td>0.7</td>
<td>12</td>
<td>0.1676</td>
</tr>
</tbody>
</table>

$xToken$ [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:

$$\frac{|\text{priceAtWithdraw} - \text{priceAtDeposit}|}{\text{priceAtDeposit}} \leq \text{tol} \quad (8)$$

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 $\text{profitAllowance} / \text{tokenVolume}$, 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.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>mint</th>
<th>burn</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>xToken</td>
<td>ethtoctr</td>
<td>miniAmi</td>
<td>sellGasSend</td>
<td>tokToRedeem</td>
</tr>
</tbody>
</table>
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 execution [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 focuses 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.

```java
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.

```java
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.