Staff Discussion Paper/Document d’analyse du personnel—2024-10

Last updated: July 9, 2024

Analysis of DeFi Oracles

by Xun Deng,[1] Sidi Mohamed Beillahi,[1] Cyrus Minwalla,[2] Han Du,[2] Andreas Veneris[1] and Fan Long[1]

[1]University of Toronto
xun.deng@mail.utoronto.ca, sm.beillahi@utoronto.ca, veneris@eecg.toronto.edu, fanl@cs.toronto.edu

[2]Information Technology Services
Bank of Canada
CMinwalla@bankofcanada.ca, HDu@bankofcanada.ca

DOI: https://doi.org/10.34989/sdp-2024-10 | **ISSN** 1914-0568

Acknowlegements

We thank ICSE 2024 anonymous reviewers for their insightful comments on an early version of this paper. This work was supported by Mitacs through the Mitacs Accelerate program.

Abstract

This paper presents OVer, a framework designed to automatically analyze the behaviour of decentralized finance (DeFi) protocols when subjected to a "skewed" oracle input. OVer firstly performs a symbolic analysis on the given contract and constructs a model of constraints. Then, the framework leverages a satisfiability modulo theory solver to identify parameters that allow its secure operation. Furthermore, guard statements can 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 used 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. Moreover, this paper delves into the design considerations of price oracles within a potential blockchain-based digital currency.

Topics: Central bank research, digital currencies and fintech, payment clearing and settlement systems

JEL codes: G, G1, G15, E4, E42, E5, E51, O, O3, O31

Résumé

Cette étude présente OVer, un cadre conçu pour analyser automatiquement le comportement des protocoles de finance décentralisée lorsqu’ils sont soumis à un oracle « biaisé ». OVer effectue d’abord une analyse symbolique d’un contrat donné et construit un modèle de contraintes. Ensuite, le cadre s’appuie sur un solveur de satisfiabilité modulo des théories pour définir les paramètres qui permettent son fonctionnement sûr. De plus, des mesures de sauvegarde peuvent être générées pour les contrats intelligents qui peuvent utiliser les valeurs de l’oracle, empêchant ainsi les attaques par manipulation de ce dernier. Les résultats empiriques montrent qu’OVer peut analyser avec succès les dix protocoles de référence sélectionnés, qui englobent une gamme variée de protocoles de finance décentralisée. En outre, l’étude montre que les paramètres actuels utilisés dans la majorité des protocoles de référence ne permettent pas de garantir la sécurité en cas de divergences importantes portées par l’oracle. Elle montre également que les mécanismes de contrôle ad hoc existants, comme l’introduction de délais, sont souvent insuffisants, voire nuisibles, pour protéger les protocoles contre de telles déviations dans le monde réel. Enfin, l’étude examine les considérations relatives à la conception des oracles de prix dans le cadre d’une éventuelle monnaie numérique basée sur la chaîne de blocs.

Sujets : Monnaies numériques et technologies financières; Recherches menées par les banques centrales; Systèmes de compensation et de règlement des paiements

Codes JEL : G, G1, G15, E4, E42, E5, E51, O, O3, O31

1 INTRODUCTION

Blockchains offer decentralized, programmable, 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 1, 2023, the total value locked (TVL) in 1,417 DeFi contracts had reached $50.15 billion (Llama Corporation n.d.(a)).

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 the *oracle values* (Adler et al. 2018). These are external values that capture the 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 application programming interfaces. Deviations in these oracle values from their true values can lead to deviations in the intended operations of the contracts (Adler et al. 2018; Cai et al. 2021). 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, automated tool for analyzing oracle deviation and verifying its impact in DeFi smart contracts. Given the source code of a smart contract protocol and the 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 one user account. Such code patterns are typically intractable for standard program analysis techniques, which would often have to make undesirable over-approximations or 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 loop computations in DeFi protocols consists of accumulators applied to map 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 (Mariano et al. 2020), the OVer algorithm adeptly manages these prevalent complexities in popular DeFi contracts.

We evaluate 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 none of seven benchmarks that have loops.

With OVer, we study the oracle deviation history in real-world blockchains. We investigate how oracle deviation would affect the behavior of popular DeFi contracts and whether existing ad-hoc mechanisms are sufficient to neutralize oracle deviations. Our results show that for six out of the seven benchmark protocols, the control mechanism was insufficient to handle oracle deviations 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 deviations. For example, to protect against potentially malicious oracle value providers, several DeFi protocols introduce delays when using oracle value inputs in their calculations (*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.

Furthermore, the blockchain is a compelling choice for implementing central bank digital currency (CBDC) systems. Ensuring the security and reliability of interactions between smart contracts and oracles is paramount for supporting financial activities on blockchain-based digital currency platforms. In our research, we conduct a thorough security analysis of seven prominent price oracle systems and simulate the impact of different time-weighted filters under volatile market conditions. Our findings provide valuable implications for designing secure oracles, enhancing the overall integrity of financial services with blockchain-based CBDCs and fostering their adoption.

In summary, this paper makes the following contributions.

- *OVer:* OVer is the first sound analysis and verification tool for analyzing oracle deviation in DeFi protocols.
- *Loop summary algorithm:* A novel loop summary algorithm is proposed to enable the analysis of sophisticated loops in DeFi smart contract source codes.
- *Results:* A systematic evaluation of OVer is presented. This is also 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 DeFi protocols against oracle deviations in the real-world.

- *Discussions:* In our study, we conduct a comparative analysis of existing price oracle solutions, benchmark diverse pricing functions and filters, and derive design considerations for blockchain-based CBDC price oracles, based on the results.

The remainder of the paper is organized as follows. Sections 2 and 3 introduce the study's 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 analyze existing price oracle solutions in Section 6. We discuss related work and threats to the validity in Section 7. And we conclude in Section 8.

2 BACKGROUND

Blockchains and smart contracts. Blockchains operate as decentralized distributed systems, offering 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 a blockchain, established to administer transaction rules integral to ledger operations. Commonly scripted in sophisticated languages such as *Solidity* (ethereum 2023), these smart contracts are later compiled into a lower-level machine language like Ethereum Virtual Machine bytecode (Buterin 2014). 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. These 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, which is contingent on the asset's volume in the liquidity pool, to ascertain an asset's price. Furthermore, the 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 blockchains 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 for 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*. An accuracy deviation emerges when a value deviates from its actual or true value, while a latency deviation is identified when an outdated value is reported, a phenomenon that can, in turn, influence accuracy. These deviations can originate from various sources such as intentional manipulation of oracles

```
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

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.

Complexity of DeFi smart contracts. Smart contracts implementing DeFi protocols, such as lending, DEXes, and derivatives (dydx-protocol 2021; aave n.d.; euler-xyz 2023a), can be complex, since they generally include loops that iterate through data structures representing various asset types or accounts managed by the protocols and calculate a sum, for example, total assets or debt. 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 (Mariano et al. 2020).

3 EXAMPLE AND OVERVIEW

We present a motivating example of applying OVer to analyze oracle deviation in *Compound* (compound-finance 2020). Figure 1 presents a simplified code snippet from *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 must 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:

1 $\sum_{a \in assets}(collFact_a * exchRt_a * p_a * cTokenBal_a) - (\sum_{a \in assets}(brwBal_a * p_a + c_a * (p_{brw} * brwAmt + collFact_r * exchRt_r * p_r))) > 0$

Figure 2: Compound analysis summary

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

where *exchRt*$_a$ is the exchange rate of the collateral asset a, p_a is the price of asset a fetched from an external oracle contract (`v.oraclePrice` at line 20), *cTokenBal*$_a$ is the balance of asset a and *collFact*$_a$ is a control variable that is smaller than one and determines 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*, 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 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 collateral, 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* (Chainlink Foundation n.d.) and the trading price of the assets in decentralized protocols such as *Uniswap* (Uniswap Labs 2023). However, the values from these sources may deviate from their ground truths. In fact, when a digital asset price is volatile, it is typically impossible to obtain an asset's fair price. For instance, if the prices in equation 1 are inaccurately reported as high, the value of the users' collateral would increase, potentially leading the protocol to execute borrowing transactions even when the users are not sufficiently collateralized.

To tackle this issue, *Compound* enforces additional margins for the positions of each collateral asset and `collFact` determines the margin sizes. *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 an additional collateral burden on users and make the protocol unattractive.

Utilizing OVer. We now show how we apply OVer to analyze *Compound* to determine the 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 the 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 of the 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`. This 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 expression are either loaded contract states (for example, `v.brwBal`) or the return values of external function calls (for example, `getUnderlyingPrice`).

Loop summarization. OVer handles the loop from lines 12 to 26 as follows. With the observation that most loops in DeFi contracts perform fold operations, particularly accumulations, OVer summarizes the loop by identifying all of the accumulations performed and replacing the loop with either one or multiple compact expression(s). By replacing the variables and the 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 behaviour 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 and b represent *CollBal*, *brwBal* and *brwAmt*, respectively. P and P_b stand for the ground truth values, while p and p_b stand for the 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 solution using satisfiability modulo theory. Finally, we pass the model to the optimizer, which iteratively calls a satisfiability modulo theory (SMT) solver to prove the constraint specified in the model above. Following this, it returns the optimal δ if one is found. Then we can insert proper `require` statements into the source code to ensure correct behaviour, for instance, by restricting the oracle deviation to be less than the value found.

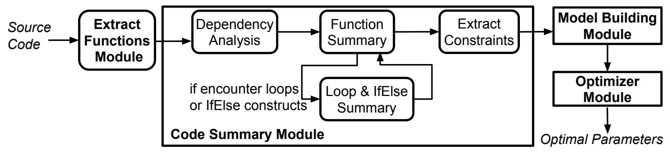

Figure 3: Overview of the proposed framework

4 DESIGN

We begin by introducing 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 the String type (line 1 in Listing 1). The contract class 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), respectively, allow a local variable to be declared and for a value to be assigned to it. The statement *load* allows a contract's state to be read.

Control-flow structures. Conditional branches are represented by the *IfThenElse* construct. A *for* loop is constituted of the loop iterator *i*, the upper bound *n* (for simplicity we omit the 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 and can only appear at the beginning in a loop body or after an *IfThenElse* construct. The *require* statement enforces smart contract constraints, and the logic behind these constraints, 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 the return values. Some function calls represent queries of the states and the oracle. In our optimization problem, we consider these 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 the 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 the *CodeSummary* module extracts concise summaries, including summaries of the loops and the conditionals, and returns a list of constraints. Next, the *BuildModel* module constructs an optimization model from the list of constraints for an SMT solver. Lastly, the *SolveOpt* module solves the optimization model, using the SMT solver.

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

```
1: procedure SummaryAnalysis($sc$)
2:     $FuncObj \leftarrow$ ExtractFunc($sc$)
3:     $ConstLst \leftarrow$ CodeSummary($FuncObj$)
4:     $M \leftarrow$ BuildModel($ConstLst$)
5:     $OptVar \leftarrow$ SolveOpt($M$)
6:     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 the loops' bodies, using the accumulation operator. To achieve 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(𝕍_1,𝕍_2) | 𝕍_1(𝕍_2) |
              𝕍_1 aop 𝕍_2 | ret(𝕍_2, 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 be either an accumulation value (*Acc*), a value ($\mathbb{V}$) or an arithmetic operation between two expressions. The $\mathbb{C}$ type can be either 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 the map $\mathbb{V}_1$ with the key $\mathbb{V}_2$. The type of $\mathbb{V}_1$ must be either an array or a 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 $\mathbb{E}$, which can be a complex mathematics formula involving multiple index and member-access operators.

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

Return values of function calls. We utilize *ret($\mathbb{V}$, i)* to indicate that $\mathbb{V}$ is the return value of a *pure* function that reads global states. When $\mathbb{V}$ is 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 the expressions and statements to include in our optimization model, we need to find which variables depend on the oracle price. To this end, we propose a set of rules *O1-O4* to infer this dependency and introduce the rule *O5* to find the guard statements that are oracle-dependent. Moreover, to compute the loop summary, we need to find which variables inside a loop body depend 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 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 that 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.

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

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

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

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

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

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

Figure 4: Function summary extraction rules

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.

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

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 $\mathbb{S}$ to map a function f to its summary. Our approach uses a ***bottom-up*** algorithm that begins from a *return* or a *require* statement. This algorithm 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 the symbol *id* with *S(f'($\mathcal{E}$))*.

The more-involved summarizations of the 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, then we halt our 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 the previous summaries computed on 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 the 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 used to summarize the effects of the *IfThenElse* construct. Similar to the loops, we enumerate the symbols in their order of dependency (line 4). We then handle the statements in reverse order and generate a summary for each symbol. To combine the summaries from both branches, we follow the *phi* instruction and take the branching 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)
```

The 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 consists 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 oracle-dependency-analysis rules *O1-O5*. The second set of constraints corresponds to the guard statements that only contain symbols that appear in the set of oracle-dependent constraints.

Algorithm 3 *IfSm* procedure. This 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 store the symbols of $\mathbb{E}$. Summaries of the symbols in M_1 and M_2 are stored in *V*.

```
1: procedure IfSm($\mathcal{E}_c, \mathcal{S}_1^*, \mathcal{S}_2^*, \Phi^*, \mathbb{E}$)
2:     $V \leftarrow \{\}$
3:     $\mathbb{E}' \leftarrow \mathbb{E}$
4:     for each $m_1 \in M_1, m_2 \in M_2$ in their order of dependency
5:       $p_t \leftarrow m_1,\ p_e \leftarrow m_2$
6:       for each $stmt_t \in reverse(\mathcal{S}_1^*), stmt_e \in reverse(\mathcal{S}_2^*)$
7:         $p_t \leftarrow$ ExtractSummary$(stmt_t, p_t)$
8:         $p_e \leftarrow$ ExtractSummary$(stmt_e, p_e)$
9:       $V[m_1] \leftarrow p_t, V[m_2] \leftarrow p_e$
10:    for each $\phi_{id} \equiv phi(id_1, id_2) \in \Phi^*$
11:      $V[id] \leftarrow V[id_1] \times Int(\mathcal{E}_c) + V[id_2] \times (1 - Int(\mathcal{E}_c))$
12:    for each $m \in M$
13:      $\mathbb{E}' \leftarrow \mathbb{E}'[m/V[m]]$
14:    return $\mathbb{E}'$
```

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

```
1: procedure BuildModel($ConstLst$)
2:     $Cv, Sd, Re, Ub \leftarrow$ ExtractVars$(ConstLst)$
3:     $Cv, Sv, Re, Gt, \Delta \leftarrow$ InitVar$(Cv, Sv, Re)$
4:     $C0, C1 \leftarrow$ InitConst$(Cv, Sv, Re, Gt, \Delta)$
5:     $C \leftarrow [C0, C1]$
6:     for each $const \in ConstLst$
7:       $[C_{Re}, C_{Gt}] \leftarrow$ ConvertZ3$(const, Re, Gt, Cv, Sv, Ub)$
8:       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 constraints. *M* has 7 parameters: Oracle price values (*Re*), ground truth price values (*Gt*), oracle deviation deltas (Δ), the DeFi protocols control variables (*Cv*) such as margin ratios, state variables (*Sv*), loop upper bounds (*Ub*) and the set of constraints extracted *C*.

To generate *M* from the guard statements, we first extract and initialize the variables (lines 2 and 3 in Algorithm 4). We also add two additional constraints, *C0* and *C1*, to all of the models (lines 4 and 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 statement, we generate two constraints, one evaluated with ground truth values, and the other, 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 behaviour as when given the ground truth price. For example, if we have the require statement *require(a > b)*, the corresponding constraint is *a > b*, and we assume that one or both of the variables *a, b* are functions of the oracle inputs. We need to prove the following:

$$a(Re) > b(Re) \Rightarrow a(Gt) > b(Gt) \tag{3}$$

$$a(Re) <= b(Re) \Rightarrow a(Gt) <= b(Gt) \tag{4}$$

Since we focus on the inputs when the transaction is not reverted, 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. This takes in a model, *M*, and returns the optimum parameters, if found.

```
1: procedure SolvOpt($M$)
2:     $ConsList \leftarrow$ simplifyConstraints($M$)
3:     while Stop condition not met
4:         $res \leftarrow$ Solv($ConsList$)
5:         $OptVar \leftarrow$ Update($res$)
6:     return $OptVar$
```

Several optimization problems 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*, we are also 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 the optimum parameters, or the procedure reaches a timeout.

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

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

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 the DeFi protocols safe under large oracle deviations?
RQ 2: Can OVer efficiently analyze the various DeFi protocols that use oracles?
RQ 3: Can OVer help developers design safe DeFi protocols that use oracles?

5.1 Implementation and benchmarks

We implement OVer based on the *Slither* static analysis tool (crytic. n.d.) with 1,160 lines of code in *Python* for Solidity-based smart contracts. To solve the optimization problems, we leverage the SMT solver, *Z3* (de Moura and Bjørner 2008). Note that the constructs of the programming language in Listing 1 that we used to present the main components of OVer's design are commonly found in other programming languages. Thus, OVer's implementation can also be extended to handle smart contracts written in other programming languages such as Vyper (Vyper Team n.d.).

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 the DeFi industry database DeFiLlama (Llama Corporation n.d.(a)) but also the protocols that fell victim to oracle manipulation attacks. To the best of our knowledge, *Aave*, *Compound*, *Solo*, *Morpho* and *Beefy* have not been victims of oracle manipulation attacks. The protocols that were victims to these 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 managers. We exclude several DeFi protocols, for example, *Inverse Finance*, *CheeseBank*, *JustLend*, *Venus*, *Benqi* and *Radiant*, which were forked from protocols in our benchmarks, for example, *Compound* and *Aave*. We also evaluate OVer on a fictional DeFi protocol (calvwang9 2022), developed to demonstrate oracle manipulation, and we call it *TestAMM*. All of our 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 to answer **RQ1**, we examine how oracle deviations impact the correctness of the 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 the Ethereum network. We gather price updates for ETH/USD, USDT/ETH, USDC/ETH and the DAI/ETH pair on *Chainlink* and the ETH/USDT pair on *Uniswap*, where USDT, USDC and DAI are stable coins that are issued in Ethereum and that stay closely one-to-one with the US dollar. We select *Chainlink* and *Uniswap* because they are very widely used oracles among the DeFi protocols (Llama Corporation n.d.(b)), as 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

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

days of ETH for the two pairs between June 2020 and September 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 a window of 155 blocks.

Table 3 shows the top five deviations found. The first and third columns give the block number when the deviation is observed on *Chainlink* and *Uniswap*, respectively. The second and fourth columns give the exact values of the 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 a failure as occurring when the user's borrowed value is greater than their collateral value. 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 oracle deviation, based on the history of these occurrences. Specifically, protocols relying on the *Chainlink* price feed, for example, *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 the price oracle and *testAMM* relies 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, for example, is that a malicious attacker could send timely transactions, during the deviation, to borrow or redeem assets with insufficient collateral, thereby 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 protocol charge fees for most operations are proportional to the transaction amount. *xToken* charges a maximum fee of 2%, while there is no deposit or withdrawal fee in *Beefy* vaults. Consequently, if we employ a 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 (yearn 2021; Yearn [yDai] Exploiter 2021).

Effect of introducing a delay. Introducing a delay is a widely recommended approach to counteracting 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 a 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 deviations due to the delay, therefore, making the underlying protocols more vulnerable. For instance, during the period from block 11541949 to block 11596096, after applying the delay method, we notice an increase in the maximum deviation from 0.0329 to 0.1525. This would make the deviation surpass the tolerable thresholds of the 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 protecting the DeFi protocols against an 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, respectively, in Table 2. Notably, both the *Compound* and *Warp* protocols share the same set of constraints for their borrowing 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 that is responsible for checking the liquidity in actions such as minting and withdrawal. As for *dForce*, *Morpho* and *TestAMM*, we analyze the safety constraint of the borrowing 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*'s compilation time in column 4 and the total execution time in column 5. Moreover, we measure the number of vector variables and other variables (scalar) in the constraints (columns 5 and 6). 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 of the protocols and their safety constraints in less than 10 seconds. We manually validate all of the generated symbolic expressions. For 8 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; for example, the majority of the execution time is consumed by Slither to parse the code and generate the AST.

5.4 Case study analysis

To answer **RQ3** and show how OVer can help developers to design safe protocols, we present case studies that apply OVer on the nine benchmark protocols. 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.

The timeouts are set to be two minutes each throughout the experiments.

Compound relies on the open price feed module to access and retrieve price information that is critical to its operations. As discussed in Section 3, the protocol implements a one-side risk control mechanism. For example, it uses a single control variable to govern its behaviour. 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 their borrowed value. When cf is greater than 1, the protocol allows under-collateralization, a situation generally considered undesirable 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, that is, 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 the *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 were to 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 the loop bounds.

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

Table 5: Compound borrowing 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 (dydxprotocol 2021) is a marginal trading protocol of *dXdY*, which uses *Chainlink* as a price oracle. A desired property in the *Solo* protocol is that for all operations, accounts remain in collateralized positions. Besides, for liquidation operations, the protocols may not want to execute unnecessary liquidations, thus verifying that the accounts being liquidated are indeed under-collateralized before proceeding with the actions. Protocol developers employ two control parameters to safeguard these operations: the margin ratio (mr) and the margin premium (mp). For liquidation to safely occur, 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 a summation and are functions of the 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 that columns 2 and 3 present the *mp* and *mr* achieved. When the bound *Ub* is 1, we achieve the margin that is set in the protocol. However, as *Ub* is increased to 2, the *mr* achieved, denoted as mr', also increases, resulting in a looser control effect. The experiment encounters a timeout when the *Ub* is 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 (dforce-network 2021) is also a pool-based lending protocol and uses *Chainlink* as a price oracle. While *dForce* also mandates over-collateralization, different from *Compound*, *dForce* designers enforce two-sided risk control, using two 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 the collateral and loan balances, and *pc* and *pb* represent the oracle prices. The constraint contains two summations, where the bounds are represented by *cl* and *bl*, respectively.

While for most assets the protocol does not use *bf* (*bf=1*), we set cf=0.5 and bf=0.7 for the purposes of our experiment. As there are two 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 where *cl > 1, bl > 1*.

xToken (sNXS 2020) serves as a liquidity manager protocol and 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, but this is ineffective when a big volume of tokens is traded. Thus, it is crucial to parameterize the variable *tol* to take the amount of tokens traded into consideration. For example, we can parameterize *tol* as $\frac{profitAllowance}{tokenVolume}$, which restricts the profits from a single transaction.

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

We use OVer to examine mint, burn, deposit and withdraw, and automatically extract the expression that approximates the price at withdrawal 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 withdrawal price is represented by the value transferred to the user divided by the token burned.

Table 8: Expressions extracted for xToken

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

TestAMM (calvwang9 2022) implements a simple lending protocol, which allows deposits of USDC and loans in ETH. It employs an AMM as a price oracle and uses a single risk control parameter named the collateralization ratio (*cr*). The simplified constraint for the borrowing action takes the form

$$amount \leq (deposits * price) * cr \tag{9}$$

The result of applying OVer with *cr* set to 0.7 is shown in Table 9. In this case, the *price* is computed using the constant product formula X * Y = K. Substituting the *price* with X/Y yields identical outcomes.

Table 9: testAMM borrow with cr = 0.7

Variables	cr′	NumVars	Time (s)
$\delta = 0.1$	0.770	3	0.2403
$\delta = 0.01$	0.710	3	0.0569
$\delta = 0.001$	0.705	3	0.0423

Aave (aave n.d.) is one of the most popular pool-based lending protocols. Chainlink is employed as the primary price oracle, with a fallback oracle serving as backup when queries to Chainlink fail. *Aave* enforces strict risk-control mechanisms using two control variables, namely, the liquidation threshold, *lth*, and the loan-to-value ratio, *ltv*. For our borrowing action experiment, we set *lth*=0.7

and *ltv*=0.5. The simplified constraints for this action are shown in Equations 10 and 11:

$$\frac{\sum_{c\in assets}(pc[c]\times cb[c]\times lth)}{\sum_{b\in assets}(pb[b]\times bb[b])}\geq 1 \tag{10}$$

$$\sum_{b\in assets}(pb[b]\times bb[b])+pa\times amt\leq\sum_{c\in assets}(pc[c]\times cb[c]\times ltv) \tag{11}$$

where *cb* and *bb* represent the collateral and loan balances, and *pc* and *pb* represent the oracle prices. *pa* and *amt* represent the price and the amount of the asset the user wishes to borrow.

The constraint for liquidation is the same as in Equation 10 except the comparator changes to less than. We initially set the loop bound to one. When we increase the loop bound to two, the number of variables in the equations increases to 11, causing z3 to time out. The experiment for the liquidation call gives a similar result. A timeout occurs when we increase the loop bound to two.

Table 10: Aave borrow with lth = 0.7, ltv = 0.5 and bound = 1

Variables	lth′	ltv′	NumVars	Time (s)
$\delta = 0.1$	0.7	0.612	7	22.2926
$\delta = 0.01$	0.7	0.511	7	2.5755
$\delta = 0.001$	0.7	0.502	7	0.2063

Table 11: Aave liquidation with lth = 0.7 and bound = 1

Variables	lth′	NumVars	Time (s)
$\delta = 0.1$	0.7	5	0.0331
$\delta = 0.01$	0.7	5	0.0321
$\delta = 0.001$	0.7	5	0.0359

Warp (warpfinance 2020) enables liquidity provider(LP) token-based borrowing and uses Uniswap V2 as a price oracle. The validation logic in the Warp protocol includes two loops, also appearing in the constraint, with their respective bounds represented by *cl* and *bl*. The control variable, namely the collateralization ratio, *cr*, is hard-coded to be two-thirds within the protocol. The experiment result is shown in Table 12. When we increase the bound *cl* to two, the number of variables reaches nine, resulting in a timeout.

Table 12: Warp borrow with cr = 2/3 and ex = 1

Variables	cr′	NumVars	Time (s)
$\delta = 0.1, cl = 1, bl = 0$	0.815	6	3.6597
$\delta = 0.01, cl = 1, bl = 0$	0.681	6	0.3280
$\delta = 0.001, cl = 1, bl = 0$	0.670	6	0.0330
$\delta = 0.1, cl = 1, bl = 1$	0.815	9	6.8087
$\delta = 0.01, cl = 1, bl = 1$	0.681	9	0.4758
$\delta = 0.001, cl = 1, bl = 1$	0.670	9	0.0467

Morpho is a lending optimizer that implements a peer-to-peer layer over pool-based protocols such as *Aave* and *Compound*. We study *Aave*'s V3-based implementation (morpho-org 2023). The simplified constraint is shown in Equation 12, where *cb* and *bb* represent the collateral and loan balances, and *pc* and *pb* represent the oracle prices. The constraint contains two summations, where the bounds are represented by *cl* and *bl*, respectively. There is only one control variable in *Morpho*, *ltv*, which is set to 0.7 in the experiment. In addition, we assume that the two bounds are equal. The result is shown in Table 13. However, a timeout occurs when we increase the bound to two.

$$\sum_{c=0}^{cl}(cb[c]*pb[c]*ltv)\geq\sum_{b=0}^{bl}(bb[b]*pc[b]) \tag{12}$$

Table 13: Morpho borrow with ltv = 0.7

Variables	ltv′	NumVars	Time (s)
$\delta = 0.1, bound = 1$	0.860	6	0.7198
$\delta = 0.01, bound = 1$	0.715	6	0.0947
$\delta = 0.001, bound = 1$	0.705	6	0.0478
$\delta = 0.1, bound = 2$	N/A	12	TO
$\delta = 0.01, bound = 2$	0.715	12	1.1773
$\delta = 0.001, bound = 2$	0.705	12	0.1006

The *Euler* protocol (euler-xyz 2023a) a pool-based lending protocol, shares the same control variables as the *dForce* protocol, but the *Euler* only has one loop in its validation logic. This protocol enforces two-sided risk control. In the experiment, we set the control variables, the collateral factor *cf* to 1 and the borrowing factor *bf* to 0.7. By fixing cf' at 1, we perform a search for the achieved value of *bf*. The results of the experiment align with those of the *dForce* protocol, as expected.

Table 14: Euler borrow with bf = 0.7, cf = 1

Variables	bf′	NumVars	Time (s)
$\delta = 0.1, bound = 1$	0.7	4	0.0312
$\delta = 0.01, bound = 1$	0.7	4	0.0311
$\delta = 0.001, bound = 1$	0.7	4	0.0311
$\delta = 0.1, bound = 2$	0.860	8	0.4572
$\delta = 0.01, bound = 2$	0.715	8	0.4570
$\delta = 0.001, bound = 2$	0.705	8	0.2674

Table 15: Expressions extracted for Beefy

Protocol	mint	burn	NumVars	Time (s)
Beefy	$\frac{amount}{shares}$	$\frac{r}{shares}$	4	0.675

Beefy is a yield aggregator protocol, and we analyze its curve-kava-3pool (Kavascan 2022). Similar to *xToken*, OVer helps analyze the mint/deposit and burn/withdraw functions and approximate the price at withdrawal and the price at deposit. The extracted expressions are shown in Table 15.

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 PRICE ORACLE DESIGN FOR THE CBDC PLATFORM

In this section, we delve into various pricing algorithms and the design of popular price oracles. Our evaluation centers around three critical attributes: accuracy, timeliness and security. Accuracy refers to the deviation of the reported value from ground truths. It measures how closely the oracle's reported price aligns with actual market values. An accurate price oracle provides reliable data for financial systems, ensuring precise calculations and informed decision making. Timeliness relates to the latency in updating the value on the blockchain. Swift and up-to-date information is crucial for real-time financial applications. A timely price oracle ensures that the most recent market data is reflected promptly, minimizing delays in transaction execution. Security considerations revolve around the resources needed to manipulate the oracle.

6.1 Security analysis of existing price oracles

6.1.1 Aggregation- and liquidity-based price oracles. We categorize price oracles based on their data sources: *aggregation-based* or *liquidity-based*. Each type has distinct trade-offs regarding accuracy, timeliness and security. An *aggregation-based* oracle gathers trading information from various platforms and reports an aggregated answer. One notable example is Chainlink. Aggregating the answers from independent nodes often yields high accuracy, timeliness and security. However, it is crucial to note that if an attacker gains control over half of the nodes, the security properties become compromised.

A *liquidity-based* oracle typically employs its own *asset pricing function* to determine the asset's price. For instance AMMs, such as Uniswap, are commonly utilized as price oracles. Due to their reliance on mathematical formulas and the number of tokens in the pool, they are susceptible to manipulation in the liquidity pool, for example, through a flash loan. Another example is seen in yield protocols, where the share price upon deposit is calculated as the ratio of the total supply and total shares of the asset. Careless implementation may allow price manipulation through re-entrance or donation-based attacks. Thus, this type of oracle often offers high timeliness and accuracy but at the cost of low security.

Both *aggregation-* and *liquidity-based* oracles tend to employ filters, such as weighted averages and medians, to remove outliers and prevent one- or multi-block price manipulations. In particular, Uniswap uses time-weighted average prices (TWAP) with arithmetic means in V2 and upgrades to geometric means in V3. Chainlink employs volume-weighted average prices (VWAP) when aggregating the trading data. Additionally, median filtering is widely used across both types of oracles. Selecting the middle value from a sorted list of prices filters out extreme outliers. Variations have been developed to further preprocess the data points and calculate a refined average. For example, Euler introduced the time-weighted-median (TWM) method.

6.1.2 Aggregation and asset pricing algorithms. Table 16 summarizes the key characteristics of widely used price aggregation algorithms and asset pricing functions. Specifically, the first column gives the name of the algorithm. The second column specifies whether the algorithm uses single or multiple data sources. The third column indicates if it is an aggregation method. The last column indicates where the algorithm is usually deployed.

$$P_{VWAP} = \frac{\sum_{i=1}^{V} v_i * p_i}{\sum_{v_i}^{V} v_i} \tag{13}$$

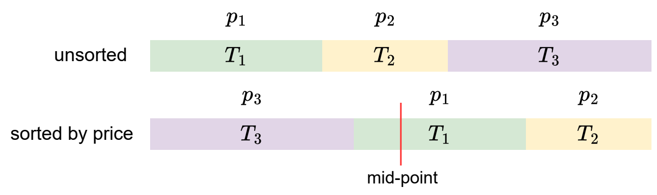

Figure 5: Illustration of the time-weighted-median algorithm

$$P_{TWAP} = \frac{\sum_{i=1}^{T} t_i * p_i}{\sum_{t_i}^{T} t_i} \tag{14}$$

The VWAP aggregates price information from multiple exchange services or nodes. In Equation 13, v_i, p_i represent the volume and price of the trade and V represents the total number of trades. The VWAP considers both the trading volume and the price, providing a robust average that reflects market dynamics. By incorporating the volume, the VWAP minimizes the impact of extreme trades (Chainlink Foundation 2023). Conversely, the TWAP and its variants, such as the exponential moving average price (EMAP), typically rely on a single source (Pyth Network 2024). Equation 14 shows the arithmetic mean version of the TWAP, where T represents the time window and p_i represents the price at time t_i. As an aggregation method, the median can be applied both to a single source in the time domain and across different sources at specific time stamps. The TWM represents a variation of the median and the time average. The TWM first sorts the prices from low to high and then selects the price at the median of the time window (euler-xyz 2023b). The algorithm is illustrated in Figure 5. To manipulate the TWM, an adversary must control the price for at least half of the time window. This effectively removes outliers over time and thereby enhances security.

If aggregation happens on-chain, the cost of gas is an important aspect to consider during implementation. While the VWAP is often implemented off-chain, the TWAP is often implemented on-chain along with the DEX protocol.

The last three rows in Table 16 present asset pricing algorithms. The implementations are on-chain, as the primary use of these algorithms is found in the AMM protocols.

$$K = \Pi_i^N x_i \tag{15}$$

$$C = \sum_i^N x_i \tag{16}$$

Although constant product AMMs (CPAMMs) are relatively straightforward to implement and have gained popularity in decentralized exchanges, they may encounter challenges in maintaining accuracy, particularly during volatile market conditions. In Equation 15, x_i represents the asset volume in the pool and K is a predetermined constant. The CPAMM model assumes that the product of the asset quantities remains constant. As trades occur, the pool rebalances automatically. The Constant SUM + Constant product (CSCP) AMM is a variation of the CPAMM, introduced by Curve for stable swap pools (Egorov 2019; Port and Tiruviluamala 2022).

This variant addresses certain limitations of the CPAMM, providing a more balanced pool and stable peg. The CSCP combines a constant sum, shown in Equation 16, and a constant product, achieving better stability for stablecoin swaps. The logarithmic market scoring rule (LMSR) has wide applications in prediction and auction markets (Hanson 2003a; Hanson 2003b). It determines the cost of providing liquidity, based on the logarithmic function. However, the LMSR can be susceptible to compromise, primarily due to its lack of sensitivity to liquidity.

It is important to note that these algorithms are building blocks of price oracles and aggregation algorithms can be combined with other pricing methods as filters. In the following section, we show how these algorithms are used in existing oracles.

Table 16: Examples of aggregation and asset pricing algorithms

Algorithm	Sources	Aggregated	On-chain/off-chain
VWAP	multiple	Yes	off-chain
MEDIAN	multiple / single	Yes	on-chain
TWAP	single	Yes	on-chain
TWM	single	Yes	on-chain
EMAP	single	Yes	on-chain
CPAMM	single DEX	No	on-chain
CSCP	single DEX	No	on-chain
LMSR	single DEX	No	on-chain

6.1.3 Existing oracle systems. Table 17 presents the algorithm(s) used by popular oracle systems and analyzes each oracle's security, timeliness and accuracy. Chainlink is a decentralized network where nodes employ off-chain VMAP algorithms to calculate average prices across multiple trading platforms. Subsequently, the data reported by these nodes is aggregated using a median approach. Chainlink has high security as attackers need to control 51% of the nodes to manipulate the price data. It also offers high timeliness and accuracy, as the price is continuously updated based on the deviation and elapsed time. However, Chainlink also faces a potential single point of failure. For instance, the Venus Protocol once suffered from a price suspension for Luna on the Chainlink platform, receiving incorrect, hard-coded price data. Uniswap is a representative protocol of TWAP-based oracles and CPAMMs. Here, we observe a trade-off between security and accuracy/timeliness. While price updates can occur after each transaction within the protocol, a lagging effect exists in the reported price, due to the use of TWAP. TWAP contributes to the security of AMM-based oracles, yet it remains feasible to manipulate the oracle through meticulously crafted transactions. Chronicle aggregates prices from trusted oracles and computes the median of the reported data. It shares similar properties with Chainlink. The difference is that Chainlink utilizes multiple layers of aggregation and Chronicle employs a single layer. Pyth employs a unique cost function that calculates a confidence-weighted median and an inversely confidence-weighted exponential moving average. While this approach effectively eliminates outliers, the accuracy and timeliness of the data may be influenced by a fixed time window, potentially underestimating confidence levels by assuming correlated errors. Two other noteworthy protocols are DIA and Euler. DIA provides a range of filters to preprocess trading data and allows for the customization of update intervals. On the other hand, Euler implements an oracle using the TWM algorithm, serving as a backup for Uniswap V3.

6.2 Simulation of TWAP and TWM in volatile periods

As discussed in the previous subsection, protocols employ various filters to enhance security, with certain systems allowing users to customize the filters. We emphasize the importance of filters in oracles with single sources, such as AMMs, which are more susceptible to manipulation. Thus our evaluation focuses on assessing the impact of TWAP and TWM on AMMs. The simulation utilizes trading data from the Uniswap ETH/USD pair during the most volatile days. We next describe the settings of our simulation.

Price of an asset. We derive the price of an asset by considering the ratio between the number of tokens in the liquidity pool. An alternative approach would involve using the trading price, which is calculated based on changes in token quantities. Additionally, we assume that the oracle reports a price once per block.

Price sampling. While it is feasible to apply medium, mean or VWAP filters to aggregate all of the trades in one block, the differences among these filters are minimal in typical scenarios. We follow Uniswap's implementation, which samples the price once per block. Specifically, we use the first update as the *reference price* in the subsequent discussions.

Simulation results. Figure 6 shows the simulation results of the price changes on May 24, 2021, from block 12494000 to block 12500999. We employ a 144-block time window, corresponding to approximately 30 minutes on the Ethereum network. The blue line represents the reference price, which reverses as a baseline. It reflects the raw price data without any additional filtering. The orange dashed line represents the TWM filter. Notably, the TWM exhibits noticeable price-level jumps, indicating its sensitivity to rapid changes. The green line depicts the TWAP with an arithmetic mean. The TWAP provides a smoother trajectory compared with the TWM, minimizing abrupt fluctuations.

Figure 7 illustrates the deviations observed across different methods, calculated as the changes between consecutive updates. Notably, similar spikes in prices are evident with no filters and the TWM. In contrast, deviations are considerably smaller when employing the TWAP method.

Table 18 summarizes the mean square error (MSE) with respect to the reference price and the realized volatility (standard deviation) of the price series. The first three rows demonstrate that employing the mean or median to aggregate trades in one block yields results that are comparable to directly sampling the price series. When we utilize the TWAP and TWM filters, the volatility of the time series decreases. Yet, this reduction in volatility comes at the cost of a lagging effect, as reflected in the MSE values.

Table 17: Analysis of popular oracles

Examples	Algorithm	Security	Timeliness	Accuracy
Chainlink	VWAP+MEDIAN	high	high	high
Uniswap V2	TWAP(arithmetic) + CPAMM	medium	medium	medium
Uniswap V3	TWAP(geometric) + CPAMM	medium	meidum	medium
Chronicle	MEDIAN(multiple oracles)	high	high	high
Pyth	CWM + ICW-EMAP	high	medium	medium
DIA	filter + MEDIAN	high	high	medium
Euler	TWM + CPAMM	high	medium	medium

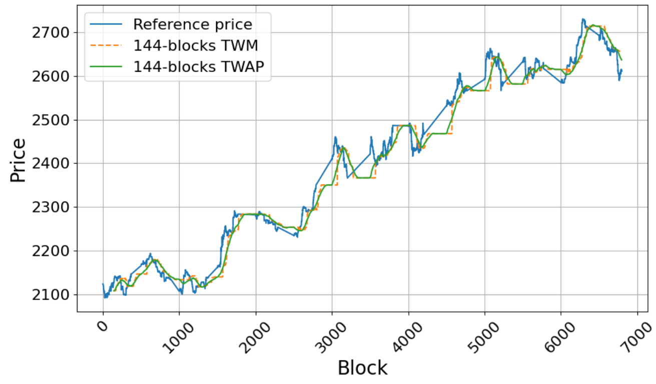

Figure 6: Reference price, TWAP and TWM on May 24th, 2021

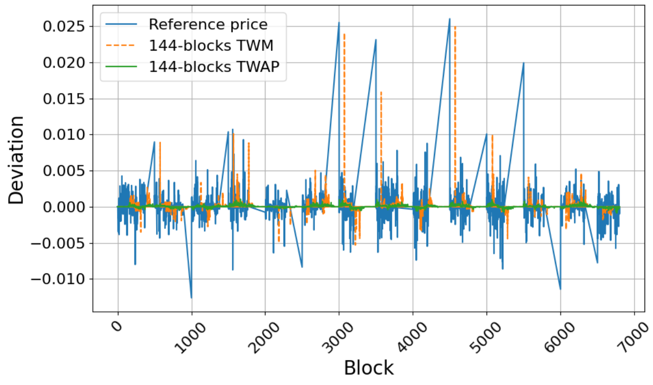

Figure 7: Deviations resulting from TWAP and TWM on May 24, 2021

We extend our simulation to more-recent blocks, specifically from blocks 18752500 to blocks 18759499 (December 10, 2023), representing normal day behaviour. Figure 8 shows the reference price and prices after applying the TWM filter and the TWAP filter. Despite a delay in price updates, the filtered values closely track the reference prices. Figure 9 shows the deviations between consecutive price updates. The TWAP exhibits the least deviation, while significant deviations persist with the TWM. Table 18 summarizes the MSE and volatility values for the different filtering methods.

Table 18: Mean square error and volatility

	May 24, 2021		Dec 10, 2023	
	MSE	Volatility	MSE	Volatility
Reference	0	192.33845	0	8.5340
Mean	0.5407	192.3212	0.00312	8.5352
Median	0.5081	192.3209	0.0032	8.5350
TWAP144	528.2182	189.8173	5.0364	8.4912
TWM144	700.3281	189.6743	6.1557	8.5915

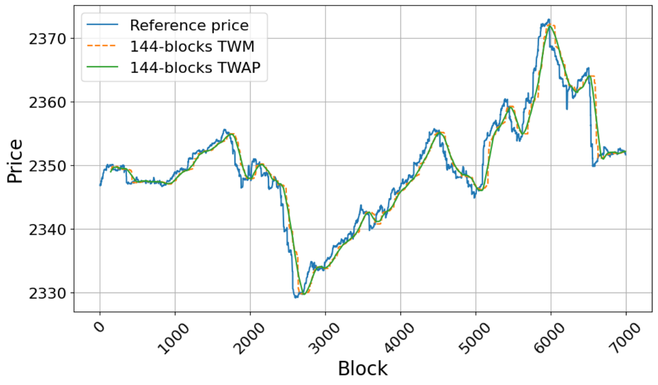

Figure 8: Reference price, TWAP and TWM on Dec 10, 2023

6.3 Oracle design for a digital currency system

We next discuss considerations when designing a price oracle for blockchain-based CBDCs. We envision two types of assets within the digital currency platform, that is, *CBDC-native* and *non-native* assets. CBDC-native assets are exclusive to the digital currency platform, and we assume the availability of exchange services, either order-book or AMM-based, for these assets on the digital currency

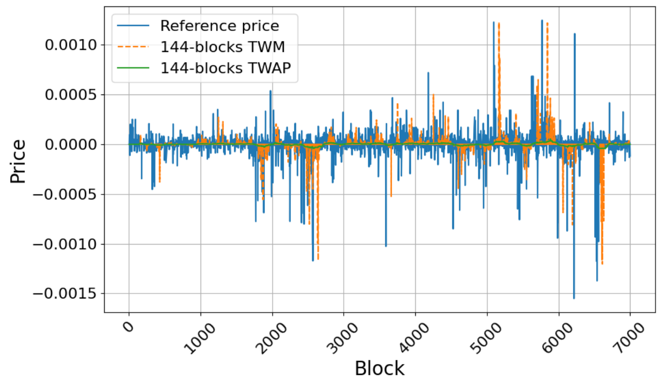

Figure 9: Deviations Resulting from TWAP and TWM on Dec 10, 2023

platform. Non-native assets are assets that exist in the outside world. Examples include fiat money and crypto assets from external blockchain networks. The system's overview is shown in Figures 10 and 11. As various services on the digital currency platform will use the price oracle, the design needs to be manipulation resistant. Configurations such as the update frequency could be customizable and may depend on the deviation and the time elapsed.

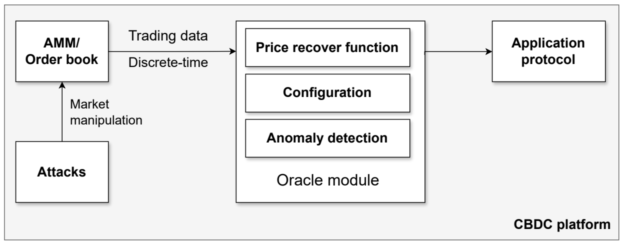

Figure 10: Oracle system overview for CBDC-native assets

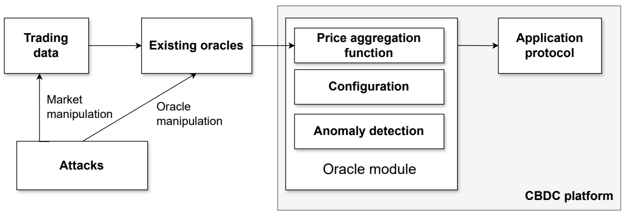

Figure 11: Oracle system overview for non-native assets

CBDC-native assets. The oracle module retrieves trading data from the exchange services on the digital currency system and recovers the asset price through the price recovery function. We believe the VWAP is a good choice for order-book-based assets, as it captures market demand and supply dynamics. Applying the VWAP with some filters will smooth the price curve and increase the cost of potential manipulation attacks. Similarly, for AMM-based assets, we propose using a TWAP filter. This approach mitigates abrupt fluctuations and provides a more stable price representation. Developing anomaly detection techniques may stop potential manipulation and enhance security. Adversaries can potentially manipulate the output of the oracle module by manipulating the AMMs or the order books. Detecting trades with huge volumes or updates significantly deviating from previous updates can help identify potential attacks. Further research is necessary to establish robust criteria for anomalies, taking into account factors such as asset type and trading frequency.

The oracle module will likely be implemented on-chain, so the cost of gas is a critical factor. The Euler's implementation of the TWM offers a more cost-effective alternative compared with the TWAP. Reducing the number of bits to represent prices contributes to this efficiency. This consideration underscores the importance of optimizing not just for accuracy but also for efficiency in the implementation.

Non-native assets. For assets that exist outside the system, the oracle is responsible for aggregating pricing information from diverse sources and bringing it into the digital currency system. Here, we assume that our oracle module aggregates the prices reported by existing oracles external to the digital currency platform. Thus, attacks on external sources may induce oracle deviations in the digital currency system. For this reason it is necessary to use the VWAP or the median to fortify the accuracy and resilience of the oracle system. These methods provide robustness against extreme outliers and ensure a more reliable representation of external asset prices. Implementing a reputation system for data providers adds an additional layer of safety. Furthermore, deploying anomaly detection mechanisms, for example, identifying abnormal price fluctuations or sudden deviations, can help prevent malicious attacks on the oracle.

7 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 (Choi et al. 2021; Jiang, Liu and Chan 2018; Nguyen et al. 2020; Shou, Tan and Sen 2023; Wang et al. 2020.), symbolic executions (Consensys 2023; Liu et al. 2020; Luu et al. 2016; Mossberg et al. 2019; Zheng, Zheng and Luo 2022) and static analysis (Feist, Grieco and Groce 2019; Kalra et al. 2018; Tikhomirov et al. 2018; Tsankov et al. 2018) 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 (Bernardi et al. 2020; Sun and Yu 2020; Tolmach et al. 2021). All of 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 (Ellul and Pace 2018; Li, Choi and Long 2020; Rodler et al. 2018). 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 et al. (2022) propose a simulation-based approach for lending protocols, searching for optimal parameters to minimize nonrepayable 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 (Chen, Beillahi and Long 2022; Deng, Zhao et al. 2023; Qin et al. 2021). Additionally, the manipulation of oracles and price manipulation attacks have been extensively discussed. For instance, the work of Mackinga, Nadahalli and Wattenhofer (2022) demonstrates the vulnerability of lending protocols that employ TWAP oracles to under-collateralized loan attacks. Xue et al. (2022) suggest monitoring token changes in liquidity pools to detect anomalous transactions and propose using front-running as a defense mechanism against such attacks. Wu et al. (2021) 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 to calculate the cost associated with initiating price manipulation attacks (Aspembitova and Bentley 2022).Wang et al. develop a tool that detects price manipulation vulnerabilities by mutating states (Wang et al. 2021). Several works focus on the design of robust oracles and proving the properties of price oracles (Dave, Sjöberg and Sun 2021). While previous research 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 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 (Mariano et al. 2020) that proposes a DSL containing map, zip and fold operations and their variants 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 that are without if branches and 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 Mariano et al. (2020), 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 the same source, *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.

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 introduce OVer, the first sound analysis tool that aids developers in constructing formal models directly from contract source code. Our findings demonstrate that OVer has the ability to analyze a broad spectrum of prevalent DeFi protocols. Intriguingly, with the assistance of OVer, we discover that many existing DeFi protocols' control mechanisms, even with default parameters, fall short in safeguarding these 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. Additionally, we conduct comprehensive assessments of widely utilized algorithms within price oracles, examining their potential integration into blockchain-based CBDC systems. Our analysis encompasses diverse considerations aimed at enhancing the precision and robustness of the oracle system, thereby securing smart contracts on blockchain-based CBDC platforms.

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 (Deng, Beillahi et al. 2023). Furthermore, this report is an extended version of a paper accepted at the International Conference on Software Engineering 2024 (Deng et al. 2024).

REFERENCES

aave. n.d. "protocol-v2." Accessed April 23, 2023. https://github.com/aave/protocol-v2/tree/master.

Adler, J., R. Berryhill, A. Veneris, Z. Poulos, N. Veira and A. 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)*, 1145–1152. Doi: 10.1109/Cybermatics_2018.2018.00207.

Aspembitova, A. T. and M. A. Bentley. 2022. "Oracles in Decentralized Finance: Attack Costs, Profits and Mitigation Measures." *Entropy* 25 (1): 60.

Bartoletti, M., J. Chiang, T. Junttila, A. L. Lafuente, M. Mirelli, and A. Vandin. 2022. "Formal Analysis of Lending Pools in Decentralized Finance." In *Proceedings of the 11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, Part III, 335–355. https://doi.org/10.1007/978-3-031-19759-8_21.

Bernardi, T., N. Dor, A. Fedotov, S. Grossman, N. Immerman, D. Jackson, A. Nutz, L. Oppenheim, O., N. Rinetzky, N. Sagiv, M., Taube, M. and Wilcox, J. R. 2020. "WIP: Finding bugs automatically in smart contracts with parameterized invariants." In *4th Stanford Blockchain Conference 2020.*

Buterin, V. 2014. "Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform." White paper.

Cai, Y., N. Irtija, E. E. Tsiropoulou and A. Veneris. 2021. "Truthful Decentralized Blockchain Oracles." *International Journal of Network Management* 32 (3): e2179.

calvwang9. 2022. "oracle-manipulation." Accessed July 15, 2023. https://github.com/calvwang9/oracle-manipulation.

Chainlink Foundation. 2023. "TWAP vs. VWAP Price Algorithms." https://chain.link/education-hub/twap-vs-vwap.

Chainlink Foundation. n.d. "ChainlinkClient API Reference." Accessed August 1. 2023. https://docs.chain.link/any-api/api-reference/.

Chen, Z., S. M. Beillahi and F. Long. 2022. "FlashSyn: Flash Loan Attack Synthesis via Counter Example Driven Approximation." arXiv preprint, arXiv:2206.10708.

Choi, J., D. Kim, S. Kim, G. Grieco, A. Groce and S. K. Cha. 2021. "SMARTIAN: Enhancing Smart Contract Fuzzing with Static and Dynamic Data-Flow Analyses." In *Proceedings—2021 36th IEEE/ACM International Conference on Automated Software Engineering*, 227–239. https://doi.org/10.1109/ASE51524.2021.9678888.

compound-finance. 2020. "compound-protocol." Accessed April 23, 2023. https://github.com/compound-finance/compound-protocol/releases/tag/v2.8.1.

Consensys. 2023. "mythril." Accessed July 6, 2023. https://github.com/Consensys/mythril.

crytic. n.d. "slither." Accessed July 28, 2023. https://github.com/crytic/slither.

Dave, K., V. Sjöberg and X. 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, edited by B. Bernardo and D. Marmsoler, 1:1–1:14. Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2021.1.

de Moura, L. and N. Bjørner. 2008. "Z3: An Efficient SMT Solver." In *Tools and Algorithms for the Construction and Analysis of Systems*, edited by C. R. Ramakrishnan and J. Rehof, 337–340. Berlin, Germany: Springer.

Deng, X., S. M. Beillahi, C. Minwalla, H. Du, A. Veneris and F. Long. 2023. "Artifact for OVer: Safeguarding DeFi Smart Contracts against Oracle Deviations." Zenodo. Accessed December 27, 2023. https://doi.org/10.5281/zenodo.10436720.

Deng, X., Z. Zhao, S. M. Beillahi, H. Du, C. Minwalla, K. Nelaturu, A. Veneris and F. Long. 2023. "A Robust Front-Running Methodology for Malicious Flash-Loan DeFi Attacks." In *2023 IEEE International Conference on Decentralized Applications and Infrastructures (DAPPS)*, 38–47. https://doi.org/10.1109/DAPPS57946.2023.00015.

Deng, X., S. M. Beillahi, C. Minwalla, H. Du, A. Veneris and F. Long. 2024. "Safeguarding DeFi Smart Contracts against Oracle Deviations." In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, Article 171, 1–12. https://doi.org/10.1145/3597503.3639225.

dforce-network. 2021. "LendingContractsV2." Accessed July 16, 2023. https://github.com/dforce-network/LendingContractsV2/tree/master/contracts.

dydxprotocol. 2021. "solo." Accessed July 18, 2023. https://github.com/dydxprotocol/solo/releases/tag/v0.41.0.

Egorov, M. 2019. "StableSwap—Efficient Mechanism for Stablecoin Liquidity." Self-published, Curve Finance. https://curve.fi/files/stableswap-paper.pdf.

Ellul, J. and G. J. Pace. 2018. "Runtime Verification of Ethereum Smart Contracts." In *2018 14th European Dependable Computing Conference*, 158–163. IEEE. DOI 10.1109/EDCC.2018.00036.

ethereum. 2023. "solidity." Accessed August 1. 2023. https://github.com/ethereum/solidity.

euler-xyz. 2023a. "euler-contracts." Accessed December 20, 2023. https://github.com/euler-xyz/euler-contracts.

euler-xyz. 2023b. "median-oracle." Accessed January 15, 2024. https://github.com/euler-xyz/median-oracle.

Feist, J., G. Grieco and A. 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)*, 8–15. DOI:10.1109/WETSEB.2019.00008.

Hanson, R. 2003a. "Combinatorial Information Market Design." *Information Systems Frontiers* 5 (1): 107–119.

Hanson, R. 2003b. “Logarithmic Markets Coring Rules for Modular Combinatorial Information Aggregation.” *The Journal of Prediction Markets* 1 (1): 3–15.

Jiang, B, Y. Liu and W. K. Chan. 2018. “ContractFuzzer: Fuzzing Smart Contracts for Vulnerability Detection.” In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 259–269. https://doi.org/10.1145/3238147.3238177.

Kalra, S., S. Goel, M. Dhawan and S. Sharma. 2018. “Zeus: Analyzing Safety of Smart Contracts.” In *Network and Distributed Systems Security (NDSS) Symposium 2018*, 1–12. http://dx.doi.org/10.14722/ndss.2018.23082.

Kavascan. 2022. “Beefy Vault Contract.” Accessed July 16, 2023. https://kavascan.com/address/0xC3821F0b56FA4F4794d5d760f94B812DE261361B?t=code.

Li, A., J. A. Choi and F. Long. 2020. “Securing Smart Contract with Runtime Validation.” In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 438–453. https://doi.org/10.1145/3385412.3385982.

Liu, Y., Y. Li, S.-W. Lin and R. 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. https://doi.org/10.1145/3368089.3409740.

Llama Corporation. n.d.(a). “DeFi Dashboard.” Accessed April 1, 2023. https://defillama.com/.

Llama Corporation. n.d.(b). “Oracles Dashboard.” Accessed April 1, 2023. https://defillama.com/oracles.

Luu, L., D.-H. Chu, H. Olickel, P. Saxena and A. Hobor. 2016. “Making Smart Contracts Smarter.” In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 254–269. https://doi.org/10.1145/2976749.2978309.

Mackinga, T., T. Nadahalli and R. 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.

Mariano, B., Y. Chen, Y. Feng, S. K. Lahiri and I. Dillig. 2020. “Demystifying Loops in Smart Contracts." In *ASE '20: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 262–274. https://doi.org/10.1145/3324884.3416626.

morpho-org. 2023. “morpho-aavev3-optimizer.” Accessed July 17, 2023. https://github.com/morpho-org/morpho-aave-v3/releases/tag/v1.0.0.

Mossberg, M., F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson and A. 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)*, 1186–1189. DOI:10.1109/ASE.2019.00133.

Nguyen, T. D., L. H. Pham, J. Sun, Y. Lin and Q. 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. https://doi.org/10.1145/3377811.3380334.

Port, A. and N. Tiruviluamala. 2022. “Mixing Constant Sum and Constant Product Market Makers.” arXiv:2203.12123 [q-fin.TR]

Pyth Network. 2024. “EMA Price Aggregation.” https://docs.pyth.network/price-feeds/how-pyth-works/ema-price-aggregation.

Qin, K., L. Zhou, B. Livshits and A. Gervais. 2021. “Attacking the DeFi Ecosystem with Flash Loans for Fun and Profit.” In *Financial Cryptography and Data Security: 25th International Conference*, Revised Selected Papers, Part 1, 3–32. https://doi.org/10.1007/978-3-662-64322-8_1.

Rodler, M., W. Li, G. O. Karame and L. Davi. 2018. “Sereum: Protecting Existing Smart Contracts Against Re-entrancy Attacks.” In *Network and Distributed Security Symposium 2019 Accepted Papers*. https://www.ndss-symposium.org/ndss-paper/sereum-protecting-existing-smart-contracts-against-re-entrancy-attacks/.

Shou, C., S. Tan and K. 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. https://doi.org/10.1145/3597926.3598059.

sNXS. 2020. “xToken Victim Contract.” Etherscan. Accessed December 28, 2022. https://etherscan.io/address/0x04bef870de607519c91d16a23434ad5745f62a63#code.

Sun, T. and W. Yu. 2020. “A Formal Verification Framework for Security Issues of Blockchain Smart Contracts.” *Electronics* 9 (2): 255. DOI:10.3390/electronics9020255.

Tikhomirov, S., E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko and Y. 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. https://dl.acm.org/doi/10.1145/3194113.3194115.

Tolmach, P., Y. Li, S.-W. Lin and Y. Liu. 2021. “Formal Analysis of Composable DeFi Protocols.” In Financial Cryptography and Data Security FC 2021 International Workshops: CoDecFin, DeFi, VOTING, and WTSC, Revised Selected Papers 25, 149–161. https://doi.org/10.1007/978-3-662-63958-0_13.

Tsankov, P., A. Dan, D. Drachsler-Cohen, A. Gervais, F. Buenzli and M. Vechev. 2018. “Securify: Practical Security Analysis of Smart Contracts.” In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 67–82. https://doi.org/10.1145/3243734.3243780.

Uniswap Labs. 2023. “Uniswap Protocol.” https://uniswap.org/.

Vyper Team. n.d. “Vyper.” Accessed July 26, 2023. https://vyper.readthedocs.io/en/stable/.

Wang, H., Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma and Y. Liu. 2020. “Oracle-Supported Dynamic Exploit Generation for Smart Contracts.” In *IEEE Transactions on Dependable and Secure Computing* 19, 1795–1809. DOI:10.1109/tdsc.2020.3037332.

Wang, S.-H., C.-C. Wu, Y.-C. Liang, L.-H. Hsieh and H.-C. Hsiao. 2021. “ProMutator: Detecting Vulnerable Price Oracles in DeFi by Mutated Transactions.” In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 380–385.

warpfinance. 2020. “Warp-Contracts.” Accessed July 16, 2023. https://github.com/warpfinance/Warp-Contracts/releases/tag/v2.0-production-contracts.

Wu, S., D. Wang, J. He, Y. Zhou, L. Wu, X. Yuan, Q. He and K. Ren. 2021. “DeFiRanger: Detecting Price Manipulation Attacks on DeFi Applications.” arXiv:2104.15068 [cs.CR]

Xue, Y., J. Fu, S. Su, Z. A. Bhuiyan, J. Qiu, H. Lu, N. Hu and Z. Tian. 2022. “Preventing Price Manipulation Attack by Front-Running.” *Advances in Artificial Intelligence and Security Communications in Computer and Information Science*: 309–322. https://doi.org/10.1007/978-3-031-06764-8_25.

yearn. 2021. “yearn-security.” Accessed July 16, 2023. https://github.com/yearn/yearn-security/blob/master/disclosures/2021-02-04.md.

Yearn (yDai) Exploiter. 2021. “Transaction Details.” Etherscans. Accessed December 28, 2022. https://etherscan.io/tx/0xf6022012b73770e7e2177129e648980a82aab555f9ac88b8a9cda3ec44b30779.

Zheng, P., Z. Zheng and X. 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. https://doi.org/10.1145/3533767.3534395.