Analysis of DeFi Oracles

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

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

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

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 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:

```solidity
function borrowAllowed(address cToken, address brw, uint brwAmt) external returns (uint ) {
...}
uint surplus = hypotheticalLiquid(brw, cToken, 0, brwAmt);
require(surplus > 0, "INSUFFICIENT_LIQUIDITY");
...
return uint(Error.NO_ERROR); }

function hypotheticalLiquid(address acct, CToken cToken, uint redTok, uint brwAmt) internal returns (uint ) {
Account.liquidityLocalVars memory v;
for (uint i = 0; i < assets .length; i++) {
CToken asset = assets [i];
,...
return v.sumColl - v.sumBrwEfct; }
```
where $\text{exchangeRate}_a$ is the exchange rate of the collateral asset $a$, $p_a$ is the price of asset $a$ fetched from an external oracle contract ($v.\text{oraclePrice}$ at line 20), $\text{TokenBalance}_a$ is the balance of asset $a$ and $\text{collFactor}_a$ is a control variable that is smaller than one and determines the enforced over-collateralization ratio for the asset. The loop also computes $v.\text{sumBrwEffect}$ as follows:

$$\sum_{a \in \text{assets}} (\text{collFactor}_a \times \text{exchangeRate}_a \times p_a \times \text{TokenBalance}_a)$$

(1)

where $\text{exchangeRate}_a$ is the exchange rate of the collateral asset $a$, $p_a$ is the price of asset $a$ fetched from an external oracle contract ($v.\text{oraclePrice}$ at line 20), $\text{TokenBalance}_a$ is the balance of asset $a$ and $\text{collFactor}_a$ is a control variable that is smaller than one and determines the enforced over-collateralization ratio for the asset.

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, $v.\text{brwBalance}$) or the return values of external function calls (for example, $\text{getUnderlyingPrice}$).

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 ($\text{collFactor}$) 0.7 and a target collateral factor ($\text{collFactor}'$) 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:

$$\min_{\delta} \delta$$

s.t. $\forall C, D, b, p, p_0, P_0 > 0, \frac{|P_1 - p_1|}{P_1} < \delta, \frac{|P_0 - p_0|}{P_0} < \delta, \text{cf} \times \sum_{i} (C_i - D_i) \times p_i - p_0 < b > 0.$

In this model, the variables $C, D$ and $b$ represent $\text{CollBalance}$, $\text{brwBalance}$ and $\text{brwAmount}$, respectively. $P$ and $P_0$ stand for the ground truth values, while $p$ and $p_0$ 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 $\delta$ if one is found. Then we can insert proper requirements into the source code to ensure correct behaviour, for instance, by restricting the oracle deviation to be less than the value found.
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 constructed from 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 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, E^*, id^*)`, where `f` identifies the function, `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.

![Figure 3: Overview of the proposed framework](image)

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.

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

**Listing 1: Simplified solidity language**

```solidity
Contract C ::= contract(id, S*, F*)
State st ::= state(ty, id)
Func F ::= func(f, id, S*)
Type ty ::= Int | Bool | Struct | Map | Array | Bytes | Address
Id id ::= String | f, i ∈ Id
Const c ::= m ∈ Int | Bool | Bytes | Address
Op ::= +, -, *, / ; bop ::= >, >=, <, <=, =
Stmt S ::= dec(ty, id) | assign(id, E) | load(id, st) | require(E) | phi(id, i, id, ..., i)
if(S) then(S') else(S')\(Φ) |
for(i, n)(S') |
call(f, E^*, id^*) | return(E')
Expr E ::= c | id | id[id] | id.id | ¬E | E1 Op E2
```

**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. 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**

```solidity
1: procedure SUMMARYANALYSIS(sc)
2: FuncObj ← EXTRACTFUNC(sc)
3: ConstLst ← CODESUMMARY(FuncObj)
4: M ← BUILDMODEL(ConstLst)
5: OptVar ← SOLVEOPT(M)
6: return OptVar
```

**Listing 2: Code summary DSL**

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.
Return values of function calls. We utilize ret(V, i) to indicate that V is the return value of a pure function that reads global states. When 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 ∪. O1: If a pure function reads from an oracle state, then the identifier it reads to is oracle-dependent. If $S = \text{call}(f, E, id)$ and Identifier(f) = oracle, where the helper function Identifier checks whether the function is annotated as an oracle state-getter, then id ∈ OD.

$$S := \text{call}(f, E, id), \text{Identifier}(f) = \text{oracle}$$

OD = OD ∪ {id, 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.

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

OD = OD ∪ {id, S} ∪ {id, S}

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

$$E = E_1 \text{ op } E_2, E_1 \in OD \text{ OR } E_2 \in OD$$

OD = OD ∪ {E}

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

$$S := \text{call}(f, E, id), f := \text{func}(f, S^{**}), E \in OD \text{ OR } S^{**} \in OD$$

OD = OD ∪ {id, S}

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

$$S := \text{require}(E), E \in OD$$

OD = OD ∪ {S}

We use $L_i$ to denote a for loop, with iterator i, and that corresponds to $S = \text{for}(i, n, S')$. $L_D$ 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.

$$E = id[E], E = i \text{ OR } E \in LD_1$$

$$LD_1 = LD_1 \cup \{E\}$$

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

$$E = E_1 \text{ op } E_2, E_1 \in LD_1 \text{ OR } E_2 \in LD_1$$

$$LD_1 = LD_1 \cup \{E\}$$

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

$$S := \text{assign}(id, E), E \in LD_1$$

$$LD_1 = LD_1 \cup \{id\}$$

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

$$S := \text{call}(f, E, id), E \in LD_1$$

$$LD_1 = LD_1 \cup \{id\}$$

4.2.2 Symbolic value extraction. We now present a set of rules to generate a function summary, shown in Figure 4. We use ExtractSummary to refer to those rules. Specifically, ExtractSummary takes a statement S and an expression E. It applies the effect of S on E and returns the updated expression E’. We use $S$ 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 E of a function f to the initially empty summary $S(f)$.

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

For a function call, call(f, E, id), we generate a summary of f(E), denoted as S(f(E)), and replace the symbol id with S(f(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 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.
We pattern match accumulation operations within \( ps \) at line 9 and check whether a \( \phi_{sm} \) statement appears in the right-hand side (rhs) precisely once. We also confirm that the rest of the expression, \( E_s \), is loop-dependent using the loop-dependency set computed at line 8. If \( E_s \) 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 \( m \) with a summary \( \nu \). 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 \( ps \) of an accumulation operation, then we halt our execution.

In Listing 3, we show an example of a nested sum and the computed complete summaries for both \( acc \) and \( acc_1 \).

### Algorithm 2 \( LpSm \) procedure

It takes loop parameters and an expression \( E \) and returns an expression \( \nu' \). \( M \) stores symbols of \( E \), ordered based on \( S' \). \( V \) stores symbols of expressions in \( M \). \( LD \) stores expressions that are loop-dependent. \( ApplyLDRules \) updates \( LD \) using loop-dependency rules.

1. \textbf{procedure} \( LpSm(s, n, S', E) \)
2. \( V \leftarrow \{ \} \)
3. \( \nu' \leftarrow E \)
4. for each \( m \in M \) in their order of dependency
5. \( p_s \leftarrow m \)
6. for each \( stmt \in \text{reverse}(S') \)
7. \( p_s \leftarrow \text{ExtractSummary}(stmt, p_s) \)
8. \( LD \leftarrow \text{ApplyLDRules}(stmt, LD) \) using the L1-L4
9. if \( p_s \equiv \phi_{sm} \land E_s \land E_s \in LD \)
10. if \( \exists \mathcal{S}_m \) depends on \( m \)
11. \textbf{return} \( \nu' \)
12. else
13. \( i \leftarrow \text{NewIndex}() \)
14. \( ps \leftarrow m + \text{sum}(E_s[i], i', i) \)
15. if \( \text{findSum} \leftarrow \text{True} \)
16. for each \( (m, v) \in V \)
17. if \( p_s \equiv \phi_{sm} \land \text{sum}(E_s[k], i) \land \forall m \in \mathcal{S}_s \land v \equiv \nu + \text{sum}(E_s[j], i) \)
18. \( ps \leftarrow ps[m[i]] \)
19. if \( p_s \equiv \phi_{sm} \land \text{sum}(E_s[k], i) \land \forall m \in \mathcal{S}_s \land v \equiv \nu + \text{sum}(E_s[j], i) \)
20. \( ps \leftarrow ps[m[i]] \)
21. for each \( (m, v) \in V \)
22. if \( p_s \equiv \phi_{sm} \land \text{sum}(E_s[k], i) \land \phi_{sm} \in v \)
23. \( V[m[i]] \leftarrow \nu + \phi_{sm}[ps[i]/i] \)
24. \( V[m[i]] \leftarrow ps \)
25. if \( \text{findSum} \)
26. \( A \leftarrow A \), \text{assign}(m_s) \)
27. for each \( m \in M \)
28. \( \nu' \leftarrow \nu' \leftarrow \text{cons}\{m[i] \}
29. \textbf{return} \( \nu' \)

### Handling conditional branches

To account for the different branches of an \( \text{IfThenElse} \) construct, our summarization includes the condition in \( \text{IfThenElse} \). Algorithm 3 describes the procedure used to summarize the effects of the \( \text{IfThenElse} \) 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 branch. To combine the summaries from both branches, we follow the \( \phi \) instruction and take the branching condition into account (line 11).

1. \textbf{procedure} \( IfSm(E', S'_1, S'_2, \Phi', \Phi, E) \)
2. \( V \leftarrow \{ \} \)
3. \( \nu' \leftarrow E \)
4. for each \( m_1 \in M_1, m_2 \in M_2 \) in their order of dependency
5. \( p_s \leftarrow m_1, p_t \leftarrow m_2 \)
6. for each \( stmt \in \text{reverse}(S'_1), stmt \in \text{reverse}(S'_2) \)
7. \( p_t \leftarrow \text{ExtractSummary}(stmt, p_t) \)
8. \( p_s \leftarrow \text{ExtractSummary}(stmt, p_s) \)
9. \( V[m_1] \leftarrow p_s, V[m_2] \leftarrow p_t \)
10. for each \( \phi_{id} \equiv \phi_{id}[id_1, id_2] \in \Phi' \)
11. \( V[id] \leftarrow V[id] \times \text{Int}(E_0) + V[id] \times (1 - \text{Int}(E_0)) \)
12. for each \( m \in M \)
13. \( \nu' \leftarrow \text{cons}\{m[V[m]] \}
14. \textbf{return} \( \nu' \)

### Algorithm 4 BuildModel procedure

This takes in a list of constraints \( \text{ConstList} \) and returns a model \( M \) for the SMT solver.

1. \textbf{procedure} \( BuildModel(\text{ConstList}) \)
2. \( C_0, S_0, R, U \leftarrow \text{ExtractVars}(\text{ConstList}) \)
3. \( C_0, S_0, R, \Gamma \leftarrow \text{InitVars}(C_0, S_0, R, \Gamma) \)
4. \( C_0, C_1 \leftarrow \text{InitConst}(C_0, S_0, R, \Gamma, \Lambda) \)
5. \( C \leftarrow [C_0, C_1] \)
6. for each \( const \in \text{ConstList} \)
7. \( C[Rc, C] \leftarrow \text{ConvertZ3}(\text{const}, R, C, S_0, U) \)
8. \( \text{APPEND}(C, [C[Rc, C]]) \)
9. \textbf{return} \( M(A, C, S_0, R, \Gamma, C, U) \)
4.3 Model generation

We build a model, \( M \), from the extracted list of constraints. \( M \) has 7 parameters: Oracle price values (\( \text{Re} \)), ground truth price values (\( \text{Gt} \)), oracle deviation deltas (\( \Delta \)), the DeFi protocols control variables (\( C_v \)) such as margin ratios, state variables (\( S_v \)), 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, \( C_0 \) and \( C_1 \), to all of the models (lines 4 and 5). \( C_0 \) states that \( S_v, \text{Re}, \text{Gt} \) are greater than zero. \( C_1 \) states that \( \text{Re} \) deviates from \( \text{Gt} \) by at most \( \Lambda \). 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 (\( \Delta \)), or control variables (\( C_v \)), 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 \( \text{require}(a > b) \), 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:

\[
\begin{align*}
\text{require}(\text{Re}) & \implies \text{require}(\text{Gt}) \\
\text{require}(\text{Re}) & \implies \text{require}(\text{Gt}) \\
\end{align*}
\]

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: \( \text{ConsList} \leftarrow \text{simplifyConstraints}(M) \)  
3: while Stop condition not met  
4: \( \text{res} \leftarrow \text{Solv}(\text{ConsList}) \)  
5: \( \text{OptVar} \leftarrow \text{Update}(\text{res}) \)  
6: return \( \text{OptVar} \)

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, \( C_v \), 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.

\[
\begin{align*}
\max \Delta \\
\text{s.t.} \; V S > 0 : \frac{P_L - P_H}{P_H} < \delta_v, \\
a(Re, So, C_v) & \implies b(Re, So, C_v) \\
a(Gt, So, C_v') & \implies b(Gt, So, C_v') \\
\end{align*}
\]

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, Morphi, Beefy and xToken. Notably, this benchmark suite contains not only widely used DeFi protocols according to the DeFi industry database DeFiLama (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, Morphi 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
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 2 shows the maximum deviation allowed by each protocol. Parameters with higher than the maximum deviation will cause a failure. For instance, for Beefy, the fee of 0.0779 is too high, while for Uniswap, the deviation of 0.02 is too low.

Table 4: Deviation limit given specific control variables

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

Table 3: Top deviations observed on Chainlink and Uniswap

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Deviation</th>
<th>Uniswap Deviation</th>
<th>Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chainlink</td>
<td>0.0393</td>
<td>10314022</td>
<td>0.4248</td>
</tr>
<tr>
<td>Chainlink</td>
<td>0.1293</td>
<td>10314022</td>
<td>0.3353</td>
</tr>
<tr>
<td>Chainlink</td>
<td>0.1260</td>
<td>10326501</td>
<td>0.2368</td>
</tr>
<tr>
<td>Chainlink</td>
<td>0.1159</td>
<td>10314022</td>
<td>0.2356</td>
</tr>
<tr>
<td>Chainlink</td>
<td>0.0994</td>
<td>10326310</td>
<td>0.1948</td>
</tr>
</tbody>
</table>

Table 4: Deviation limit given specific control variables

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

Answer to RQ1: We surprisingly found that the default control parameters of the investigated protocols are not enough to protect these protocols against 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 a 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 countering 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

<table>
<thead>
<tr>
<th>Variables</th>
<th>cf</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ = 0.1, bound = 1</td>
<td>0.8600</td>
<td>5</td>
<td>4.5486</td>
</tr>
<tr>
<td>δ = 0.01, bound = 1</td>
<td>0.7200</td>
<td>5</td>
<td>0.1194</td>
</tr>
<tr>
<td>δ = 0.001, bound = 1</td>
<td>0.7100</td>
<td>5</td>
<td>0.0794</td>
</tr>
<tr>
<td>δ = 0.1, bound = 2</td>
<td>NA</td>
<td>9</td>
<td>TO</td>
</tr>
<tr>
<td>δ = 0.01, bound = 2</td>
<td>0.7150</td>
<td>9</td>
<td>0.3535</td>
</tr>
<tr>
<td>δ = 0.001, bound = 2</td>
<td>0.7050</td>
<td>9</td>
<td>0.2316</td>
</tr>
</tbody>
</table>

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:

\[
\frac{splyVal}{(1 - mp)} < brwVal \ast (1 + mp) \ast (1 + mr)
\]
where \(splaVal\) 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 \(mr^\prime\), 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 \(mr = 0.1\)

<table>
<thead>
<tr>
<th>Variables</th>
<th>(mp)</th>
<th>(mr)</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1, bound = 1)</td>
<td>0.15</td>
<td>0.10</td>
<td>5</td>
<td>0.0280</td>
</tr>
<tr>
<td>(\delta = 0.01, bound = 1)</td>
<td>0.15</td>
<td>0.10</td>
<td>5</td>
<td>0.0281</td>
</tr>
</tbody>
</table>

\[ \delta = 0.001, bound = 1 \]

| \(\delta = 0.1, bound = 2\) | 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:

\[
\begin{align*}
\text{bf} \times \sum_{c=0}^{\text{cl}} (\text{cb}[c] + \text{pc}[c]) > \sum_{b=0}^{\text{bl}} (\text{bb}[b] + \text{pb}[b]) \times \frac{\text{bf}}{\text{cb}}
\end{align*}
\]

where \(\text{cb}\) and \(\text{bb}\) represent the collateral and loan balances, and \(\text{pc}\) and \(\text{pb}\) represent the oracle prices. The constraint contains two summations, where the bounds are represented by \(\text{cl}\) and \(\text{bl}\), respectively.

While for most assets the protocol does not use \(bf\) (\(bf=0\)), we set \(bf=0.5\) 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 \(cf^\prime\) and \(bf^\prime\) 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 transaction. As \(k\)ens traded into consideration. For example, we can parameterize \(tol\) as \(\frac{\text{profitAllowance}}{\text{tokenVolume}}\), which restricts the profits from a single transaction.

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 user divided by the token burned. Similarly, the withdrawal price is represented by the value transferred to the user divided by the token burned.

### Table 7: \(dForce\) borrow with \(cf = 0.5\) and \(bf = 0.7\)

<table>
<thead>
<tr>
<th>Variables</th>
<th>(cf)</th>
<th>(bf)</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1, cl = 1, bl = 0)</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0264</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 1, bl = 0)</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0265</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 1, bl = 0)</td>
<td>0.5</td>
<td>0.7</td>
<td>3</td>
<td>0.0263</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 1, bl = 1)</td>
<td>0.5</td>
<td>0.950</td>
<td>6</td>
<td>3.8585</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 1, bl = 1)</td>
<td>0.5</td>
<td>0.7150</td>
<td>6</td>
<td>0.3996</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 1, bl = 1)</td>
<td>0.5</td>
<td>0.7020</td>
<td>6</td>
<td>0.1091</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 1, bl = 2)</td>
<td>0.6120</td>
<td>0.7</td>
<td>6</td>
<td>2.7778</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 1, bl = 2)</td>
<td>0.5110</td>
<td>0.7</td>
<td>6</td>
<td>0.3082</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 1, bl = 2)</td>
<td>0.5020</td>
<td>0.7</td>
<td>6</td>
<td>0.0892</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 2, bl = 1)</td>
<td>0.5</td>
<td>0.8560</td>
<td>9</td>
<td>7.1560</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 2, bl = 1)</td>
<td>0.5</td>
<td>0.7150</td>
<td>9</td>
<td>0.4700</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 2, bl = 1)</td>
<td>0.5</td>
<td>0.7020</td>
<td>9</td>
<td>0.0875</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 2, bl = 2)</td>
<td>0.6120</td>
<td>0.7</td>
<td>12</td>
<td>9.0215</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 2, bl = 2)</td>
<td>0.5110</td>
<td>0.7</td>
<td>12</td>
<td>0.3842</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 2, bl = 2)</td>
<td>0.5020</td>
<td>0.7</td>
<td>12</td>
<td>0.1010</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 2, bl = 2)</td>
<td>0.6115</td>
<td>0.7</td>
<td>12</td>
<td>1.8996</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 2, bl = 2)</td>
<td>0.5105</td>
<td>0.7</td>
<td>12</td>
<td>0.6305</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 2, bl = 2)</td>
<td>0.5015</td>
<td>0.7</td>
<td>12</td>
<td>0.1676</td>
</tr>
</tbody>
</table>

### Table 8: Expressions extracted for \(xToken\)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>mint</th>
<th>burn</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(xToken)</td>
<td>(\text{etherCon} = \text{mint})</td>
<td>(\text{burn} = \text{burn})</td>
<td>4</td>
<td>1.7247</td>
</tr>
</tbody>
</table>

\(TestAMM\) (calvwan9 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

\[
\text{amount} \leq (\text{deposits} \times \text{price}) \times \text{cr}
\]

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 \(X \times Y = K\). Substituting the \(price\) with \(X/Y\) yields identical outcomes.

### Table 9: \(testAMM\) borrow with \(cr = 0.7\)

<table>
<thead>
<tr>
<th>Variables</th>
<th>(cr)</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1)</td>
<td>0.770</td>
<td>3</td>
<td>0.2403</td>
</tr>
<tr>
<td>(\delta = 0.01)</td>
<td>0.710</td>
<td>3</td>
<td>0.0569</td>
</tr>
<tr>
<td>(\delta = 0.001)</td>
<td>0.705</td>
<td>3</td>
<td>0.0423</td>
</tr>
</tbody>
</table>

\(Aave\) (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:

\[
\sum_{cassets} (pc[c] \times cb[c] \times lth) \geq 1
\]

(10)

\[
\sum_{bassets} (pb[b] \times bb[b]) + pa \times amt \leq \sum_{cassets} (pc[c] \times cb[c] \times ltv)
\]

(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 \(z_3\) to time out. The experiment for the liquidation call gives a similar result. A timeout occurs when we increase the loop bound to two.

\[
\sum_{c=0}^{cl} (cb[c] + pb[c] \times ltv) \geq \sum_{b=0}^{bl} (bb[b] + pc[b])
\]

(12)

The experiment for the liquidation call gives a similar result. A timeout occurs when we increase the loop bound to two.

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 10: Aave borrow with \(lth = 0.7\), \(ltv = 0.5\) and bound = 1**

<table>
<thead>
<tr>
<th>Variables</th>
<th>(lth')</th>
<th>(ltv')</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1)</td>
<td>0.7</td>
<td>0.612</td>
<td>7</td>
<td>22.2926</td>
</tr>
<tr>
<td>(\delta = 0.01)</td>
<td>0.7</td>
<td>0.511</td>
<td>7</td>
<td>2.5755</td>
</tr>
<tr>
<td>(\delta = 0.001)</td>
<td>0.7</td>
<td>0.502</td>
<td>7</td>
<td>0.2063</td>
</tr>
</tbody>
</table>

**Table 11: Aave liquidation with \(lth = 0.7\) and bound = 1**

<table>
<thead>
<tr>
<th>Variables</th>
<th>(lth')</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1)</td>
<td>0.7</td>
<td>5</td>
<td>0.0331</td>
</tr>
<tr>
<td>(\delta = 0.01)</td>
<td>0.7</td>
<td>5</td>
<td>0.0321</td>
</tr>
<tr>
<td>(\delta = 0.001)</td>
<td>0.7</td>
<td>5</td>
<td>0.0359</td>
</tr>
</tbody>
</table>

Warps (warpsfinance 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 \(ex = 1\)**

<table>
<thead>
<tr>
<th>Variables</th>
<th>(cr')</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1, cl = 1, bl = 0)</td>
<td>0.815</td>
<td>6</td>
<td>3.6597</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 1, bl = 0)</td>
<td>0.681</td>
<td>6</td>
<td>0.3280</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 1, bl = 0)</td>
<td>0.670</td>
<td>6</td>
<td>0.0330</td>
</tr>
<tr>
<td>(\delta = 0.1, cl = 1, bl = 1)</td>
<td>0.815</td>
<td>9</td>
<td>6.8087</td>
</tr>
<tr>
<td>(\delta = 0.01, cl = 1, bl = 1)</td>
<td>0.681</td>
<td>9</td>
<td>0.4758</td>
</tr>
<tr>
<td>(\delta = 0.001, cl = 1, bl = 1)</td>
<td>0.670</td>
<td>9</td>
<td>0.0467</td>
</tr>
</tbody>
</table>

**Table 13: Morpho borrow with \(ltv = 0.7\)**

<table>
<thead>
<tr>
<th>Variables</th>
<th>(ltv')</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1, bound = 1)</td>
<td>0.860</td>
<td>6</td>
<td>0.7198</td>
</tr>
<tr>
<td>(\delta = 0.01, bound = 1)</td>
<td>0.715</td>
<td>6</td>
<td>0.0947</td>
</tr>
<tr>
<td>(\delta = 0.001, bound = 1)</td>
<td>0.705</td>
<td>6</td>
<td>0.0478</td>
</tr>
<tr>
<td>(\delta = 0.1, bound = 2)</td>
<td>N/A</td>
<td>12</td>
<td>TO</td>
</tr>
<tr>
<td>(\delta = 0.01, bound = 2)</td>
<td>0.715</td>
<td>8</td>
<td>1.1773</td>
</tr>
<tr>
<td>(\delta = 0.001, bound = 2)</td>
<td>0.705</td>
<td>12</td>
<td>0.1006</td>
</tr>
</tbody>
</table>

**Table 14: Euler borrow with \(bf = 0.7\), \(cf = 1\)**

<table>
<thead>
<tr>
<th>Variables</th>
<th>(bf')</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\delta = 0.1, bound = 1)</td>
<td>0.7</td>
<td>4</td>
<td>0.0312</td>
</tr>
<tr>
<td>(\delta = 0.01, bound = 1)</td>
<td>0.7</td>
<td>4</td>
<td>0.0311</td>
</tr>
<tr>
<td>(\delta = 0.001, bound = 1)</td>
<td>0.7</td>
<td>4</td>
<td>0.0311</td>
</tr>
<tr>
<td>(\delta = 0.1, bound = 2)</td>
<td>0.860</td>
<td>8</td>
<td>0.4572</td>
</tr>
<tr>
<td>(\delta = 0.01, bound = 2)</td>
<td>0.715</td>
<td>8</td>
<td>0.4570</td>
</tr>
<tr>
<td>(\delta = 0.001, bound = 2)</td>
<td>0.705</td>
<td>8</td>
<td>0.2674</td>
</tr>
</tbody>
</table>

**Table 15: Expressions extracted for Beefy**

<table>
<thead>
<tr>
<th>Protocol</th>
<th>mint</th>
<th>burn</th>
<th>NumVars</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Beefy</td>
<td>(\text{shares})</td>
<td>(\text{shares})</td>
<td>4</td>
<td>0.675</td>
</tr>
</tbody>
</table>

**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 in yield curves, where the price share 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-based 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 (TAPW) 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.

![Figure 5: Illustration of the time-weighted-median algorithm](image)

$$P_{TWAP} = \frac{\sum_{i=1}^{T} t_i * p_i}{\sum_{i=1}^{T} t_i}$$ (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 (EMA), 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 (Egorov 2019; Port and Tiruviluamala 2022). 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.

$$K = \Pi_{i=1}^{N} x_i$$ (15)

$$C = \sum_{i}^{N} x_i$$ (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 pre-determined 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 CPMM, 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>
<thead>
<tr>
<th>Algorithm</th>
<th>Sources</th>
<th>Aggregated</th>
<th>On-chain/off-chain</th>
</tr>
</thead>
<tbody>
<tr>
<td>VWAP</td>
<td>multiple</td>
<td>Aggregated</td>
<td>Yes</td>
</tr>
<tr>
<td>MEDIAN</td>
<td>multiple / single</td>
<td>Aggregated</td>
<td>Yes</td>
</tr>
<tr>
<td>TWAP</td>
<td>single</td>
<td>Aggregated</td>
<td>Yes</td>
</tr>
<tr>
<td>TWM</td>
<td>single</td>
<td>Aggregated</td>
<td>Yes</td>
</tr>
<tr>
<td>EMAP</td>
<td>single</td>
<td>Aggregated</td>
<td>Yes</td>
</tr>
<tr>
<td>CPAMM</td>
<td>single DEX</td>
<td>Aggregated</td>
<td>No</td>
</tr>
<tr>
<td>CSCP</td>
<td>single DEX</td>
<td>Aggregated</td>
<td>No</td>
</tr>
<tr>
<td>LMSR</td>
<td>single DEX</td>
<td>Aggregated</td>
<td>No</td>
</tr>
</tbody>
</table>

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 DLA and Euler. DLA 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

<table>
<thead>
<tr>
<th>Examples</th>
<th>Algorithm</th>
<th>Security</th>
<th>Timeliness</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chainlink</td>
<td>VWAP+MEDIAN</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Uniswap V2</td>
<td>TWAP(arithmetic) + CPAMM</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Uniswap V3</td>
<td>TWAP(geometric) + CPAMM</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>Chronicle</td>
<td>MEDIAN(multiple oracles)</td>
<td>high</td>
<td>high</td>
<td>high</td>
</tr>
<tr>
<td>Pyth</td>
<td>CWM + ICW-EMAP</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
<tr>
<td>DIA</td>
<td>filter + MEDIAN</td>
<td>high</td>
<td>high</td>
<td>medium</td>
</tr>
<tr>
<td>Euler</td>
<td>TWM + CPAMM</td>
<td>high</td>
<td>medium</td>
<td>medium</td>
</tr>
</tbody>
</table>

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

![Figure 6: Reference price, TWAP and TWM on May 24th, 2021](image1)

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

<table>
<thead>
<tr>
<th></th>
<th>May 24, 2021</th>
<th>Dec 10, 2023</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MSE Volatility</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reference</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mean</td>
<td>0.5407</td>
<td>0.0032</td>
</tr>
<tr>
<td>Median</td>
<td>0.5081</td>
<td>0.0032</td>
</tr>
<tr>
<td>TWAP144</td>
<td>192.3212</td>
<td>5.0364</td>
</tr>
<tr>
<td>TWAP144</td>
<td>192.3209</td>
<td>5.0364</td>
</tr>
<tr>
<td>TWM144</td>
<td>189.8173</td>
<td>8.4912</td>
</tr>
</tbody>
</table>

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

![Figure 8: Reference price, TWAP and TWM on Dec 10, 2023](image2)

### 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
Existing oracles Oracle Order book Trading attacks data world. Examples include fiat money and crypto assets from external sources and may depend on the deviation and the time elapsed.

We propose the first sound analysis tool to analyze oracle deviation and may depend on the deviation and the time elapsed.

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.

The oracle module will likely be implemented on-chain, so the cost of gas is a critical factor. The Euler’s implementation of the TWAP offers a more cost-effective alternative compared with the TWM. 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 requirements...
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.

```solidity
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.

```solidity
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


