Towards Finding Accounting Errors in Smart Contracts

Anonymous Author(s)

ABSTRACT

Bugs in smart contracts may have devastating effects as they tend to cause financial loss. According to a recent study, accounting bugs are the most common kind of bugs in smart contracts that are beyond automated tools during pre-deployment auditing. The reason lies in that these bugs are usually in the core business logic and hence contract-specific. They are analogous to functional bugs in traditional software, which are largely beyond automated bug finding tools whose effectiveness hinges on uniform and machine checkable characteristics of bugs. It was also reported that accounting bugs are the second-most difficult to find through manual auditing, due to the need of understanding underlying business models. We observe that a large part of business logic in smart contracts can be modeled by a few primitive operations like those in a bank, such as deposit, withdraw, loan, and pay-off, or by their combinations. The properties of these operations can be clearly defined and checked by an abstract type system that models high-order information such as token units, scaling factors, and financial types. We hence develop a novel type propagation and checking system with the aim of identifying accounting bugs. Our evaluation on a large set of 57 existing accounting bugs in 29 real-world projects shows that 58% of the accounting bugs are type errors. Our system catches 87.9% of these type errors. In addition, applying our technique to auditing a large project in a very recent auditing contest has yielded the identification of 6 zero-day accounting bugs with 4 leading to direct fund loss.

1 INTRODUCTION

Blockchains and cryptocurrencies have become an integral part of our economy. As of the writing of this paper, the global market cap for cryptocurrencies reaches $1.19 Trillion USD, with the top two blockchains being Bitcoin and Ethereum. An important kind of blockchain-based applications are smart contracts, which can encompass a wide range of services, from banks to gaming platforms and marketplaces. Smart contracts follow the DeFi, or *decentralized finance* principle. Unlike centralized systems, such as federal banks, smart contracts operate in a decentralized manner without a single controlling authority, rendering many novel financial applications. Similar to traditional software, smart contracts are developed by programmers and inevitably have bugs. The lucrative value of exploiting these bugs has made smart contracts one of the most popular targets of many malicious actors. As of Q2 of 2023, $300 million USD were exploited from 212 security incidents, suggesting that each exploit costed an average of $1.5 million USD.

Therefore, there is a pressing need to develop techniques to find smart contract bugs. Existing techniques can be roughly classified into four categories: static analysis, fuzzers, symbolic execution, and verification. Static analysis toolsanalyze source code without actually running the code. They usually transform smart contracts to various intermediate representations and then search for certain bug patterns. Fuzzers run contracts against a large number of inputs and transactions sequences. Symbolic execution analyzes all possible program paths of a smart contract by performing symbolic computation instead of concrete execution. Verification tools leverage formal methods to check smart contracts against formal specifications. These approaches have demonstrated great effectiveness in identifying a broad range of issues. Some bugs such as *reentrancy* and integer *overflow* can hardly survive these tools. However, most automatic techniques rely on application agnostic oracles, meaning that bugs need to be clearly defined without considering application specific semantics. Such oracles may be difficult to acquire for certain kinds of bugs. Verification tools are capable of detecting a wide spectrum of bugs including those application specific. However, they need the developers to provide application specifications, which may entail substantial manual efforts. As a result, there are still a large number of bugs that are beyond existing tools, evidenced by the growing number of attacks.

According to a recent study by Zhang et al. [1] on over 500 exploitable bugs (bugs that can lead to direct fund loss) from 119 real-world smart contract projects, 80% of exploitable bugs are *machine unauditable bugs* (MUBs), meaning that they fall outside of the scope of existing automatic tools. Among them, *financial bugs*, which are incorrect implementations of underlying contract business models, are the most common type of MUBs in projects before deployment and also the second hardest to find in manual auditing, due to the need of understanding the most complex parts of contracts, namely, the business logics. On the other hand, their impact can be devastating. An example would be the Uranium Finance Exploit. Due to two extra zeros in an interest calculation, the contract was exploited for $57 million USD. The bug survived multiple rounds of manual auditing (by experts).

In this paper, I develop a type-checking tool for accounting bugs in smart contracts. My insight is that *although financial bugs reside in complex business logic and seemingly lack an application-agnostic oracle, many manifest themselves as abstract type violations. Abstract type inference* is a technique that can be traced back to the 70's in the last century. It aims to abstract higher level semantic information such as physical units (e.g., seconds and meters) than those denoted by primitive types in programming languages such as integers and strings. As such, type systems can be enhanced to check a much richer set of properties, such as physical unit consistency in robotic systems. I further observe that *although smart contracts have sophisticated business models, their basic operations are still analogous to those in a simple bank system*, such as deposit, withdraw, and loan. I hence devise an abstract type system based on these operations that can infer and check abstract types. In particular, I model and infer three facets of each variable, which are: *token unit* indicating the kind of currency denoted by the variable (analogous to USD in real life), *scaling factor* that denotes how much the variable has been scaled in order to simulate floating point computation that is not supported in smart contract programming languages, and *financial meaning*, e.g., if the variable denotes an interest or a debt. With the rich types, I can check a large set of properties that shall be uniformly true for various business models, using

```
1  contract Pools{
2  ...
3  function addLiquidity(address base, address token, address
       member) external returns(uint liquidity) {
4      uint addedBase = getAddedAmount(base, ...);
5      uint addedToken = getAddedAmount(token, ...);
6      liquidity = calcLiquidityUnits( addedBase, totalBase,
            addedToken, totalToken, totalLiquidity);
7      liquidity[...][member] += liquidity;
8      totalBase += addedBase;    ...
9  }
10 function calcLiquidityUnits(uint b, uint B, uint t, uint T,
       uint P) external view returns (uint){
11     uint part1 = (t * B);
12     uint part2 = (T * b);
13     uint part3 = (T * B) * 2;
14     uint _units = (((P × part1) + part2) / part3);
15     return (_units) / one;
16 }
```

Figure 1: Buggy Code from *Vader* [5]

type rules. For instance, values of different token units cannot be added or subtracted together, similar to how lengths of meters and inches cannot be added together; amounts scaled by different factors should not be compared; interest should not be subtracted from debt but rather adds to it. To use my tool, the user annotates a few global variables and function parameters with their abstract types. The annotations are limited and usually clear from project description and even variable names. Then, my technique automatically infers the abstract types for other variables and performs type checking.

I implement a prototype ScType based on Slither [2]. I evaluate the system on 50 real-world contracts from [1] and Code4Rena [3]. It finds 31 bugs with 87.9% recall and 73.8% precision. In contrast, the state-of-the-art tools could only find 5 of the bugs. The paper has been accepted to ICSE 2024 (a prestigious research conference in Software Engineering) to be held in Portugal. My system can be downloaded and tested at [4].

2 MOTIVATION

We use two real-world accounting bugs to explain the inadequacy of existing techniques and illustrate our method.

Example I (Lending Contract Vader). Figure 1 contains code detailing two functions included within the *Pools* contract from the *Vader* project [5]. They have been shortened for demonstrative purposes. Vader is a lending project that enables the pooling of funds and offers borrowing functionalities. Users can participate by purchasing shares from a liquidity pool and utilize these shares as collateral to access borrowed funds, thereby increasing the pool's returns. Within the project, The Pools contract stores functions related the movement of liquidity in the pool.

The function `addLiquidity()` in Figure 1 converts a contract's recently added *base* currency and *token* currency into liquidity, and adds the liquidity to a certain `member`'s account. In the contract, the base currency is the default currency of the Vader project, while the token currency is a special currency used by the liquidity pool, representing a share of the pool. The amount of recently added base currency is computed on line 4 as `addedBase`, and the amount of recently added token is computed on line 5 as `addedToken`. The equivalent amount of liquidity is calculated through the `calcLiquidityUnits()` function call on line 6, in which `totalBase`, `totalToken`, and `totalLiquidity` denote the total amount of base currency, total token currency, and total liquidity in the pool. The resulting liquidity is then added to the `member` account on line 7 and the total base currency is updated on line 8.

The bug occurs within the function `calcLiquidityUnits()`. Associating the variables on line 6 to the formal arguments on line 10, we have that `b` is the recently added base currency, `B` is the total base currency, `t` is the recently added token currency, `T` is the total token currency, and `P` is the current liquidity. The function attempts to convert the base and token currencies to liquidity on lines 11-15 through the following equation: $(P * (t * B) + (T * b))/(T * B * 2)$, which is incorrect, with the correct equation being: $P * ((t * B) + (T * b))/(T * B * 2)$. The understanding of the exact math is unnecessary, and therefore more details are excluded. As a consequence of this bug, all conversions of currencies to liquidity are incorrect, losing the funds of both the contract and the users. This bug was ranked as *High Risk* on Code4rena, the highest severity on the platform.

```
1  contract LibBalances{
2  ...
3  function applyTrade(Position position,  Trade trade, uint256
       feeRate) internal pure returns ... {
4      int256 signedAmount = trade.amount;
5      int256 signedPrice = trade.price;
6      int256 quoteChange = signedAmount * signedPrice;
7      int256 fee = getFee(trade.amount, trade.price, feeRate);
8
9      int256 newQuote = 0;
10     int256 newBase = 0;
11     if (trade.side == LONG) {
12         newBase = position.base + signedAmount;
13         newQuote = position.quote - quoteChange + fee;
14     } else if (trade.side == SHORT) {
15         newBase = position.base - signedAmount;
16         newQuote = position.quote + quoteChange - fee;
17     }
18     ...
19 }
```

Figure 2: Buggy Code from *Tracer* [6]

Example II (Trading Contract Tracer). Figure 2 depicts a function included within the *LibBalances* library from *Tracer* [6], which is a *derivative* smart contract designed to enable users to trade in *perpetual markets* [7]. A derivative contract has its functionalities based upon derivatives, namely, financial agreements derived from an underlying asset or financial market. Examples of derivatives include stocks, options, and futures. In perpetual markets, users can place long and short trade orders aiming on buying or selling *base* tokens using *quote* tokens, respectively. Users going long earn money when the price of the base token increases, while users going short earn money when the price of the base token decreases. Every user possesses a *position* that stores the amount of base and quote tokens that the user has. The LibBalances contract is a library within the Tracer project that provides basic functionalities in its perpetual market. In particular, the function `applyTrade()` is used to finalize both long and short trades. The amount of base token associated with the trade is defined as `signedAmount` on line 4, and the price of the base token is defined in the following line 5 as `signedPrice`. The amount and price of the base token are used to calculate the equivalent amount of quote token on line 6 as `quoteChange`. Then a fee for the transaction is calculated through the `getFee()` function call on line 7.

The short trade handling from line 14 to 16 is correct, and depicts the user selling the `signedAmount` in base token and earning the `quoteChange` in quote token, minus the fee by the contract. On the other hand, the long trade handling on lines 11 to 13 is incorrect, specifically on line 13. Instead of losing the fee meant for the contract, the long trade user gains the fee instead. A likely explanation for this bug is that the developer confused the contract fee, which is charged to the user for each transaction, with the *funding fee* [8] in perpetual contracts, which is a fee paid from long traders to short traders periodically when the base token price increases, and from short traders to long traders when the base token price decreases, as an incentive for traders. The consequence of this bug is that users entering long trades would gain additional quote tokens on the contract's loss. This bug was also ranked as High Risk on Code4rena.

Existing Techniques Are Insufficient. As discussed in section 1, existing techniques such as static analysis, fuzzing, and symbolic execution require application agnostic oracles, like those used in finding reentrancy bugs. Reentrancy occurs when a victim contract makes a call to an external function before updating state variables, potentially allowing the external function to call the victim contract again. This leads to a loop which can cause unexpected behaviors such as fund draining. It can be detected by finding call cycles using various analysis: static, dynamic, and symbolic. Such an oracle does not have to model application specific behaviors. However, the previous two bugs reside in the core business logic specific to the contracts. For example, the first bug is coupled with the obscure math in function `calcLiquidityUnits()` and the second bug seems to require understanding perpetual trading and the unique design of position of the Tracer project. They hence cannot be detected by the automated tools we have tried (see section 4).

Our Technique. To address the aforementioned limitations, We propose a novel static analysis tool for detecting accounting bugs by considering inherent financial meanings of each variable. We introduce the concept of *extended type*, which denotes information such as token unit, scaling factor, and financial type for each variable and enables type propagation. This extended type information is used to check the correctness and consistency of all operations within the contract.

We demonstrate the usage of our tool here, starting with the bug in Figure 1 regarding the function `addLiquidity()`. Through type inference and propagation, ScType determines the token/currency units of `addedBase` on line 4, `addedToken` on line 5, and `totalLiquidity` on line 5, as t_{base} (meaning the token denoted by the address `base`), t_{token}, and $e(t_{base}, t_{token})$ denoting a composite expression involving t_{base} and t_{token}. More details of such inference can be found in subsection 3.4. In addition, due to line 8, `totalBase` has a token unit of t_{base}. Similarly, `totalToken` has a unit of t_{token}. On line 6 where the function `calcLiquidityUnits()` is invoked, our tool can infer that the parameters have token units of: $\{t_{base}, t_{base}, t_{token}, t_{token}, t_e = e(t_{base}, t_{token})\}$. This information is then propagated to the `calcLiquidityUnits()` function body starting on line 10. Hence on line 11, $\texttt{part1} = \texttt{t} * \texttt{B}$, and therefore has unit $[t_{base} * t_{token}]$. On line 12, $\texttt{part2} = \texttt{T} * \texttt{b}$, and has unit $[t_{base} * t_{token}]$. On line 13, $\texttt{part3} = \texttt{T} * \texttt{B} * 2$, and therefore has unit $[t_{base} * t_{token}]$ as well. However, the issue arises on line 14 when the calculation of $\texttt{\_units} = (((\texttt{P} \times \texttt{part1}) + \texttt{part2})/\texttt{part3})$ is performed. This calculation is split into three separate steps: the first calculation being $\texttt{TMP\_0} = \texttt{P} \times \texttt{part1}$, the second calculation being $\texttt{TMP\_1} = \texttt{TMP\_0} + \texttt{part2}$, and finally, $\texttt{\_units} = \texttt{TMP\_1}/\texttt{part3}$. According to the first calculation, `TMP_0` has unit $[t_{base} * t_{token} * t_e]$, so during the second calculation when `TMP_0` is added to `part2`, which has unit $[t_{base} * t_{token}]$, there is a type mismatch, and thus our tool reports an error. In contrast, the fixed version `P*((t*B) + (T*b))/(T*B*2)` can be type-checked.

For the bug in Figure 2, the parameters of `applyTrade()` have the following initial extended type information (from its documentation): `trade.amount` is a *Balance*, denoting the amount some account owns, `trade.price` is a *Price*, denoting trading price between two products, `feeRate` is a *Fee*, denoting charge to an account by the contract, and `position.quote` is a *Balance*. The types of other variables are then inferred and checked. On line 4, the financial meaning of `signedAmount` is determined as *Balance* due to the assignment. On line 6, `quoteChange` is typed as *Balance* similarly. On line 7, `fee` is typed to *Fee* by inter-procedural analysis. Then on line 13, `position.quote - quoteChange` is typed to *Balance* as it is the difference of two balances. However, the whole expression `position.quote - quoteChange + fee` cannot be typed as *Balance* + *Fee* is illegal. This is because fee is a charge and should never be added to a balance. In contrast, line 16 can be typed.

Table 1: Popular DeFi Project Categories

Project Types	Brief Summary	Typical Operations
Yield & Yield Aggregator	Yield projects allow users to deposit funds to be used as collateral in other projects, where the earnings are returned.	`stake, withdraw, draw, invest, reinvest`
Dexes	Exchange projects allow users to trade one currency for another, or for shares in a pool.	`swap, add/remove liquidity`
Lending	Lending projects allow users to borrow currency in exchange for a collateral.	`lend, repossess, repay loan`
Services	Service projects facilitate services or functions such as games and wallets.	`deposit, withdraw, buy, sell`
Derivatives	Derivative projects are based upon simulating real-world derivatives such as stocks, options, and futures.	`trade, liquidate, withdraw, deposit`

3 DESIGN

3.1 A Conceptual DeFi Model

We studied a large number of DeFi projects (i.e. a total of 113) collected in [1]. The majority of such projects have already been deployed in the real-world, and there are many of which are fairly complicated, e.g., Tigris [9] and Biconomy [10]. We have a key observation: *many of these projects can be considered as mutations of a bank*. Banks are the oldest financial institution, and their key functionality is to collect liquidity from deposits, distribute the liquidity through loans, collect interest (and possibly collateral as well through liquidation) from those loans, and remit the profits back to the deposits. Modern DeFi projects often have fairly sophisticated business models (e.g., derivative contracts). However, their essence is still *collecting unused liquidity and re-distributing for profits*. We hence propose a bank-like conceptual model to describe some fundamental operations of DeFi projects, allowing us to derive a set of

```
1  contract DeFiModel {
2    IERC20 public T; //contract token
3    IERC20 public T0; //asset token
4    uint public fee_rate;
5    uint public interest_rate;
6    uint public earning;
7    uint public totalSupply;
8    unit public totalDebt;
9    unit public price;
10   mapping(address => uint) public balance;
11   mapping(address => uint) public debt;
12   mapping(address => uint) public collateral;
13
14   function deposit(uint amount) public { //deposit T0 for T
15       unit share= swap_T0_4_T(amount);
16       uint fee= update_fee(fee_ratio, totalSupply, totalDebt);
17       balance[msg.sender]+=share;
18       balance[msg.sender]-=fee;
19       totalSupply+=share-fee;
20       earning+=fee;
21  }
22  function withdraw(uint share) public {
23       ... //fee computation
24       balance[msg.sender]-=share;
25       balance[msg.sender]-=fee;
26       earning+=fee;
27       totalSupply-=share+fee;
28       return swap_T_4_T0(share);
29   }
30   function accounting() public {
31       uint dividend = calc_dividend(earning, totalSupply,
             balance[msg.sender]);
32       earning-=dividend;
33       balance[msg.sender]+=dividend;
34       uint interest = calc_interest(debt[msg.sender]);
35       debt[msg.sender]+=interest;
36       earning+=interest;
37   }
38   function swap_T0_4_T(uint amount) public {
39       price=IERC20.balanceof(T)/IERC20.balanceOf(T0);// T0*T=k
40       return amount * price;
41   }
42   function loan(uint share, uint collateral) public {
43       ... //fee computation
44       new_collateral=collateral[msg.sender]+collateral;
45       new_debt = debt[msg.sender]+ share + fee;
46       if (new_debt*100>=new_collateral*75) return;
47       collateral[msg.sender] = new_collateral;
48       debt[msg.sender] = new_debt;
49       totalSupply- = share;
50       totalDebt + = share + fee;
51       earning+=fee;
52   }
53   function payoff (uint share) public {
54       ... //fee computation
55       debt[msg.sender]- = share-fee;
56       totalDebt - =share-fee;
57       totalSupply + = share-fee;
58       earning+=fee;
59   }
60   function liquidate (address account) require owner public {
61       if (debt[account]*100<collateral*75) return;
62       totalSupply+= debt[account];
63       earning+=collateral[account]-debt[account];
64       totalDebt-= debt[account];
65       debt[account] = 0;  collateral[account] = 0;
66   }
67 }
```

Figure 3: Bank-like Conceptual Model for DeFi Contracts

key properties. In the following, we first describe the conceptual model and then explain how existing DeFi projects can be considered as instantiations of the model. To avoid additional denotation overhead, the model is composed using a Solidity-like language in Figure 3. It is worthy noting that the code is not intended to be complete or sound, but rather provides a vehicle for the later discussions of financial properties.

State Variables. The global variables in lines 2 - 12 denote a number of key book-keepings that a bank has to maintain. Later in this section, we will show that they become the extended types in our system. In particular, `T` on line 2 is the base token of the bank, denoting the currency of bank, similar to the base currency in the Pools contract in Figure 1, while `T0` on line 3 represents a universally utilized currency, anaglogous to USDC or ETH in smart contracts and USD in real life. Variable `fee_rate` is a ratio used in fee calculation. Usually, fee charged to the customer that initiates a transaction is calculated as a percentage of the transaction. `Interest_rate` is the ratio used to calculate interest of debt. Line 6 defines the bank's earnings, which is typically incremented through fee and interest accmulations. Variable `totalSupply` denotes the total amount of asset in the contract-specific token `T`. Similarly, `totalDebt` on line 8 represents the total amount of debt in `T` token that users have accrued. On line 9, `price` represents a conversion rate between `T0` and `T`. On lines 10-12, we define a set of mappings that are used to store information specific to each user, including `balance`, `debt`, and `collateral`.

Deposit. Lines 14 - 21 denote how a customer deposits asset, which may be loaned to others to harvest interest. Note that `msg.sender` is a standard Solidity term that denotes the user, namely, the sender of the transaction message (i.e., function call). It is usually called with a deposit amount in a universal currency (i.e., `amount` in `T0`), analogous to the user depositing US dollars. The amount is first converted to a share of the overall asset in the bank on line 15, in the bank's currency `T`. Intuitively, it denotes what portion of the bank's overall asset is owned by the user. On line 16, `fee_ratio` and then `fee` are updated (typically based on the utilization of asset). Then the user balance, total asset, and earnings are updated. Note that the fee is taken from the user and saved to the bank. As we will show later, many operations in smart contracts share a similar nature to the deposit function, although their implementations may be orders of magnitude more complex. Hence, if we can type variables in smart contracts to balance, fee, total supply etc., like the variables in our function. We can check properties such as *fee should be taken from balance*, an invariant across implementations.

Withdraw. Lines 22 - 29 define how a user withdraws assets. It is almost symmetric to `deposit()`. Note that the fee is taken from the user's balance.

Accounting. Lines 30 - 37 denote the internal regular accounting of the bank, determining how balances earn dividends and debts increase with accrued interest. In particular, `dividend` calculated on line 31 denotes the distribution of bank's earnings to the user (lines 32 and 33). `Interest` calculated on line 34 increases the user's debt on line 35 and the bank's earnings. Note that the functions `calc_dividend()` and `calc_interest()` (including function `update_fee()` on line 16) are not defined as they denote bank-specific protocols. When we instantiate the conceptual model to various smart contracts, different contracts have their own innovative and project-specific definitions of those functions.

Swap. Lines 38 - 41 show how an amount of the universal coin `T0` is converted to the base token `T`. First, the price of `T` in terms of `T0` must be computed. In finance, a rule is typically followed to determine the price of an asset x when trading it with another asset y. That is, the product of total amounts of x and y on market remains unchanged by the trade [11]. To ensure this invariant, the price of trading x for y is the total amount of y divided by the total amount of x. Intuitively, if x is traded for y, x amount decreases and y amount increases, x shall become more expensive, and vice versa. Hence, the price of token `T` in terms of `T0` is total `T` divides total `T0`, demonstrated on line 39.

Loan. Lines 42 - 52 show how a user initiates a loan provided an amount of collateral. The loan amount `share` (in `T`) and the collateral amount `collateral` are given on line 42. Line 46 checks if the total debt is currently lower than 75% of the total collateral. If not, the user is not allowed to borrow. If the loan is granted, the user's collateral and debt are updated on lines 47-48. The `totalSupply` of the `T` token is decreased by the loaned amount on line 49, and the total debt `totalDebt` is increased by the loaned amount and the fee on line 50. Finally, the earnings are incremented on line 51.

Payoff. Paying off debt is largely symmetric to taking a loan, except that it does not need to check the health of account. Note that the fee is added to the debt (the same as in `loan()`).

Liquidate. Lines 60 - 66 show how an account's collateral can be liquidated if the debt is not paid off in time. The `require owner` modifier on line 61 means that only the bank owner can perform this operation. During liquidation, the health of the account is first checked on line 61 to decide if it should be liquidated. If so, the collateral is split to two parts, the first part paying off the debt and going to the total supply, and the second part goes to the bank's earnings. Then the debts and collateral are reset, as referenced on lines 64 and 65, respectively.

3.2 Instantiation of the Conceptual Model to DeFi Contracts

In the following section, we reason how our model can be instantiated to various kinds of smart contracts. According to Defillama [12], a DeFi analytics platform, there are 13 different types of smart contracts. We take the 5 most popular project types (by [13]), namely, *yield and yield aggregators*, *lending*, *dexes*, *services*, and *derivatives*, and discuss how our model can be used to model their basic functionalities. The contract categories and their typical operations are listed in Table 1.

Yield and Yield Aggregators. These projects allow users to stake funds into smart contracts, and then use the aggregated funds to generate yield as profit. They rely on *strategies*, which are automated investment strategies to earn rewards, incentives, or interest in other smart contracts. One example of a yield strategy would be contributing funds to a liquidity pool in another smart contract and receiving fees from transactions involving that pool. Once the yield strategy ends, the yield is then made available to withdraw, with any unclaimed yield being used to reinvest in the user's account. Some examples of yield and yield aggregators include *Convex Finance* (TVL[1] $3.7b) and *Sushi BentoBox* (TVL $76.2m).

Typical functions within these projects include: *stake*, *withdraw*, *invest*, and *reinvest*. Observe that there are almost direct mappings of these operations to those in the bank-like model. For example, *stake* corresponds to `deposit()`, and *invest* and *reinvest* are just special forms of withdraws as they entail reducing the user's balance and sending the reduced amount out (e.g., to a strategy account). As such, we can check behavior correctness by making sure the bookkeepings in those functions follow a similar fashion to those inside `deposit()` and `withdraw()` in Figure 3. The Vader project in section 2 is such an example. Function `addLiquidity()` in Figure 1 corresponds to `deposit()` in Figure 3. Observe that the `base` and `token` in the former correspond to `T0` in the latter (as they are the tokens the user deposits) and the `liquidity` token corresponds to `T`. The invocation to `calcLiquidity()` on line 6 in Figure 1 corresponds to the swap function call on line 15 in Figure 3. As such, we can check balances are correctly updated and fee is properly charged. We want to point out that the above discussion is conceptual, and ScType does not require explicitly constructing such correspondences. Instead, they are implicitly encoded by our extended types. More will be discussed in our type system section.

Dexes. These projects host an exchange market between two currencies. Dexes support two types of users: liquidity *suppliers* and *traders*. Suppliers deposit amounts of both token types in order to grow the contract's supply pool, receiving a portion of the trading fees as income. Traders deposit one type of token into the pool, and receive the corresponding amount of the other token, minus some fee. Examples of deployed dex projects include Uniswap (TVL $3.83b) and Balancer (TVL $1.023b). Typical functions in Dexes include *swap*, *addLiquidity*, and *removeLiquidity*. The last two correspond to `deposit()` and `withdraw()` of our model, and *swap* is equivalent to depositing in one token, swapping to another token using a function similar to `swap_T0_4_T()` in our model, and then withdrawing in the later token.

Lending. Lending projects allow users to loan money, placing a collateral at stake. These projects are directly comparable to banks. Some of the most well known lending projects include AAVE (TVL $5.867b) and Compound Finance (TVL $2.282b). Lending projects support functions such as *lend*, *liquidate*, and *repay*.

Services. These projects facilitate some services, such as a wallet and a game. They regulate the backend flow of funds of their corresponding applications. Popular service projects include Instadapp (TVL $2.082b) and DefiSaver (TVL $103m). Typical functions of service projects include: *deposit*, *withdraw*, *buy*, and *sell*. They can be expressed with operations in our bank model.

Derivatives. Derivative contracts (e.g., futures and options) allow users to speculate on price movement, interest rates, and other financial variables without directly owning the underlying asset. Some popular derivative projects include GMX (TVL $583.22m) and dYdX (TVL $357.21m). Their typical functions include: *trade*, *bid*, *liquidate*, *withdraw*, and *deposit*. A trade function, like `applyTrade()` in Figure 2, can be modeled by depositing one token, swapping to another token, and then withdrawing the later token. As such, checking the property of fee in this procedure, namely, *fee should always be at the cost of user*, regardless buy, sell, deposit, or withdraw, identifies the bug in Figure 2.

[1] TVL, or Total Value Locked, represents the amount of capital in a smart contract.

```
1  usdcAmount = USDC.balance();
2  scaledUSDCAmount = USDC.balance() * 10^12;
3  totalAmount = usdcAmount + scaledUSDCAmount;
```

Figure 4: Bank Like DeFi Model

<*Program*>	P ::= S
<*Statement*>	S ::= $S_1;S_2$ | $x := v$ | $x := y$ | $x := a$ | $x := y$ **op** z | $x :=$ **IERC20**(y_1).**balanceOf**(y_2) | **IERC20**(y_1).**transfer**(y_2, x) | **if**(x) $S1$; **else** $S2$; | **while**(x) S;
<*Var*> x, y, z	<*Value*> $v \in \{1, 5, 10^{18}, ...\}$ <*Address*> a
<*Comparison*> $\bowtie \in \{>, <, ==, != , ...\}$	<*Binop*> $op \in \{+, \times, \bowtie ...\}$

Figure 5: Language

3.3 Token Unit and Scaling Factor

Besides the financial meanings, our type system also captures the implicit token units and scaling factors for individual variables.

Token Units. On the blockchain, the majority of currencies used are tokens. Tokens have monetary value, and can be exchanged for real world currencies such as USD. Tokens most commonly used include USDC (USD Coin) and ETH (Ethereum). DeFi projects utilize these tokens, or implement their own tokens in order to supply liquidity and fuel their economies. In Solidity, these tokens are handled as integers, despite potentially being of different currencies. This is because as of current, programming languages lack built-in models for handling (monetary) units. For example, amounts of USDC and ETH are both represented using the same primitive integer type in Solidity. This may potentially allow vulnerabilities to arise when such amounts are incorrectly handled. Just like how meters and feet cannot be directly compared, even though they are both measurements of length, directly comparing two amounts of different token units should not be allowed either. To prevent such problems, each variable has its token unit in our type system.

Scaling Factor. In Solidity, using a *scaling factor* is a common technique to handle decimal numbers without using floating-point arithmetic, which is not supported in the language. A scaling factor is a power of 10 that is multiplied to an actual token amount. Many tokens have a default scaling factor; the USDC token has a factor of 10^6. This means 1 USDC (equivalent to 1 USD in real life) is internally denoted by a value of 10^6. The WETH token has a scaling factor of 10^{18}. Hence, in order to compute the price between USDC and WETH, USDC needs to be scaled by a factor of 10^{12} for the scaling factors to be equal. However, similar to token units, the onus of using scaling factors properly is completely on developers, who likely make mistakes such as forgetting to multiply/divide by the right scaling factor. We demonstrate this in Figure 4. An amount of USDC, `usdcAmount` is computed on line 1. Recall that the default scaling factor of USDC is 10^6. On line 2, a variable `scaledUSDCAmount` is set to the USDC amount scaled by a factor of 10^{12}, making the factor of `scaledUSDCAmount` 10^{18}. Finally, these two amounts are added on line 3. This should not be allowed, since the two tokens have different scaling factors. However, Solidity compiles and runs the code without issue. Therefore we model scaling factors explicitly in our type system to prevent such errors.

<*Extended Type*>	τ ::=< f, s, u, a >
<*Financial Type*>	$f \in$ {`RawBal`, `AccBal`, `NetBal`, `T-Supply`, `Fee`, `Debt`,`Interest`, `Dividend`, `Price`, `-`, ...}
<*Scaling factor*>	$s \in \mathbb{Z} \cup \{-\}$
<*Token Unit*>	$u := u \times u \mid u \div u \mid t_a \mid t_{this} \mid -$

Figure 6: Extended Types

3.4 Type System

Definitions. Figure 5 presents a language to facilitate discussion. Although ScType supports the complex syntax of Solidity, we use a simplified language for discussion. In particular, we represent variables of all types as *Var*. Only variables with primitive types of integers and addresses will be typed by ScType, since other primitive types such as boolean and string do not provide meaningful information for our purpose. We also define a selected set of statements within *Statement*. These statements are where most of the type checking and inference take place. The first three statements are assignments: $x := v$, $x := a$, and $x := y$ representing *Value*, *Address*, and *Variable* assignments, respectively. Following are binary operation statement $x := y$ op z, and two primitive function calls, namely, $x :=$ **IERC20**(y_1).**balanceOf**(y_2) and **IERC20**(y_1).**transfer**(y_2, x). The first one type-casts y_1 to an ERC20 token, intuitively some kind of currency, and then retrieves the balance of an account denoted by y_2, which contains an address. The second one transfers x amount of y_1 token from the user that initiates the transaction (i.e., `msg.sender`) to an address denoted by y_2. We model these functions as they directly disclose token units and financial types. Finally, we include statements representing conditionals and loops. Although our typing system handles other statements such as function calls, they are handled in a standard way, e.g., propagating types through parameters, and hence elided.

Extended Types. Figure 6 represents the types that we have developed for our system. In ScType, each variable is typed with a tuple of four: *Financial Type*, *Scaling Factor*, *Token Unit*, and *Address*. Financial type represents the monetary implication of the variable. Following the bank-like model in Figure 3, we list part of the supported financial types: raw balance (`RawBal`) denoting balance before accrual, balance after fee charged (`NetBal`), balance after dividend accrual (`AccBal`), total supply (`T-Supply`), transaction fee (`Fee`), debt, interest, and exchange price. The symbol '-' denotes *not-applicable*, meaning the variable has no monetary implication. We elide some supported types such as collateral for discussion simplicity. Scaling factor represents the exponent of 10 that the variable is scaled by. Token unit represents the token unit of the variable. Token units are expressions of either an existing currency denoted by address a, represented by t_a, the currency of the current contract, represented by t_{this}, and their product or ratio, which are typically used in price computation. In contrast, sum and difference (of different token units) are not legitimate. The address aspect of the type denotes the address that a variable may hold.

Type System. Our type system is based on single-static-assignment representations by Slither [2] and flow-sensitive. In other words, different left-hand-side appearances of a variable are renamed and hence typed separately. In addition, it is field-sensitive and context-sensitive, although not directly reflected in our later discussion of type rules. It types individual public/external functions in a

$$(R_1)\quad \frac{}{x := a\ :\ \langle -, -, -, a\rangle} \qquad (R_2)\quad \frac{}{x := v\ :\ \langle -, getScaleV(v), -, -\rangle}$$

$$(R_3)\quad \frac{y : \tau}{x := y\ :\ \tau}$$

$$(R_4)\quad \frac{y_1 : \tau_1 \quad y_2 : \tau_2 \quad \tau_1.u = \tau_2.u \quad \tau_1.s = \tau_2.s}{x := y_1 + y_2\ :\ \langle \tau_1.f \oplus \tau_2.f, \tau_1.s, \tau_1.u, -\rangle}$$

$$(R_5)\quad \frac{y_1 : \tau_1 \quad y_2 : \tau_2 \quad \tau_1.u = \tau_2.u \quad \tau_1.s = \tau_2.s}{x := y_1 - y_2\ :\ \langle \tau_1.f \ominus \tau_2.f, \tau_1.s, \tau_1.u, -\rangle}$$

$$(R_6)\quad \frac{y_1 : \tau_1 \quad y_2 : \tau_2}{x := y_1 \times y_2 : \langle \tau_1.f \otimes \tau_2.f, \tau_1.s + \tau_2.s, \tau_1.u \times \tau_2.u, -\rangle}$$

$$(R_7)\quad \frac{y_1 : \tau_1 \quad y_2 : \tau_2}{x := y_1 \div y_2 : \langle \tau_1.f \oslash \tau_2.f, \tau_1.s - \tau_2.s, \tau_1.u \div \tau_2.u, -\rangle}$$

$$(R_8)\quad \frac{y_1 : \langle -, -, -, a_1\rangle \quad y_2 : \langle -, -, -, a_2\rangle}{x := \textbf{IERC20}(y_1).\textbf{balanceOf}(y_2)\ :\ \langle \texttt{RawBal}, getScaleA(a_1), t_{a_1}, -\rangle}$$

$$(R_9)\quad \frac{y_1 : \tau_1 \quad y_2 : \tau_2 \quad \tau_1.u = \tau_2.u \quad \tau_1.s = \tau_2.s}{x := y_1 \bowtie y_2 : \langle -, -, -, -\rangle}$$

Figure 7: Type Rules

⊖	RawBal	NetBal	AccBal	T-Supply	Fee	Debt	Dividend
RawBal	RawBal	✗	✗	T-Supply	✗	Debt	✗
NetBal	✗	NetBal	✗	T-Supply	✗	Debt	✗
AccBal	✗	✗	AccBal	T-Supply	✗	Debt	✗
T-Supply	✗	✗	✗	T-Supply	✗	✗	✗
Fee	NetBal	✗	AccBal	✗	Fee	✗	Dividend
Debt	RawBal	NetBal	AccBal	✗	✗	Debt	✗
Dividend	✗	✗	✗	T-Supply	Fee	Debt	Dividend

Table 2: Definition of Operator ⊖ (top_row ⊖ left_column)

$$getScaleV(v) = \begin{cases} n & v = 10^n \\ - & \text{otherwise} \end{cases}$$

$getScaleA(a)$ = dictionary lookup of token a's scaling factor

Figure 8: Helper functions used in type rules

contract one-by-one. Typing one such function entails typing all the function directly/indirectly invoked as well. Some global variables and function parameters may not have their types automatically inferred, usually when such information is only implicitly assumed as preconditions. In such cases, ScType prompts the user for their types. We call such user provided information *type annotations*, which are typical in static analysis and symbolic analysis. They can be extracted from documentation, comments, and even variable names. In section 4, we will show such manual efforts are limited. In the following discussion, we assume type annotations are in place for simplicity.

Figure 7 presents our rules for type inference and checking. Within our rules, $y : \tau$ represents variable y having an extended type τ. A statement $x := ... : \tau$ means that τ is the resulting type of the statement that will be propagated to x. Special operations $\oplus$, $\ominus$, $\otimes$, and $\oslash$ represent the resulting financial meanings of addition, subtraction, multiplication, and division operations, respectively. We have included the definition of $\ominus$ in Table 2 as a reference. The definitions of remaining operators have been excluded for space, and can be found in our supplementary material.

Rule R_1 specifies that when assigning an address to a variable, the corresponding extended type τ tracks the address, which is later propagated to other places/variables through copy statements. Rule R_2 demonstrates that when assigning a value to a variable, τ records the scaling factor of the value, obtained through the helper function **getScaleV()** in Figure 8. Rule R_3 specifies that in a copy statement, the left-hand-side variable inherits its type from the right-hand-side. Rules R4-R5 specify the rules for addition and subtraction. They are in a similar form. For example, rule R_5 specifies when a variable y_1 typed to τ_1 is subtracted by y_2 typed to τ_2, their token units and scaling factors must be the same, and their financial types must be legitimate for subtraction as well (according to Table 2). In Table 2, the top row represents the minuend, the left column represents the subtrahend, and the intersection cell represents the result. A legal subtraction, such as `RawBal - Fee` yields `NetBal` (row 6 and column 2), which cannot be further subtracted by a fee (row 6 and column 3), indicated by the ✗. The intuition of the table can be derived from our DeFi model. For example, `Debt-Fee` is not allowed as fee shall increase debt, not decrease (see `loan()` in Figure 3). Some may wonder why we allow `RawBal-Debt` because paying off debt should be in the form of `Debt-RawBal`. The reason is that many DeFi projects allow users to over-pay their debts with the extra going to their balances. The extra is computed by `RawBal-Debt`. Rules R_8 and R_9 are for multiplication and division. For these two, we do not check consistencies of token units and scaling factors as multiplication and division of different tokens (with different scaling factors) are often necessary in computing trading/swapping price (see `swap_T0_4_T`() in Figure 3). The definitions of $\otimes$ and $\oslash$ are in our supplementary material. Specifically, we only allow multiplication/division within the same type of balance, not across. For example, `RawBal/RawBal` is allowed but `RawBal/NetBal` is not, as the ratio of the latter serves no purpose. Rule R_8 specifies the inference rule for **balanceOf**(...). The conditions are that both y_1 and y_2 must be of the address type. The resulting variable x has the type <`RawBal`, $getScaleA(a_1)$, t_{a_1}, ->, representing that it is a raw balance, has a scaling factor that is looked up from a dictionary, which is easy to construct with one-time effort as there are only a few default scaling factors for popular tokens, and the token unit t_{a_1}. The rule for **transfer**(...) is similar and elided. Rule R_9 specifies that for comparison operations, the token units and scaling factors must be consistent. The resulting type is a null type, since *boolean* values do not have financial meaning. The rules for conditionals and loops are standard and elided.

Examples. Recall the Vader bug in Figure 1. Variables `addedBase` and `addedToken` on lines 4 and 5 are typed to <RawBal,0,t_{base},-> and <`RawBal`, 0, t_{token}, ->, respectively, through the function calls of `getAddedAmount(...)`, which is a wrapper of `balanceOf(...)`. In other words, the two are raw balances with different token units. The addition on line 8 allows typing `totalBase` to token unit t_{base}. Similarly, `totalToken` has unit t_{token} and `totalLiquidity` has unit t_e, which is an expression of t_{base} and t_{token}. These 5 variables are the parameters passed to the `calcLiquidityUnits()` function. Through rule R_6 for multiplication, variables `part1`, `part2`, and `part3` all have type <`RawBal` , 0, $t_{token} \times t_{base}$,->. ScType reports an error during the 3-part calculation of `unit` on line 14. Specifically, the first part, $\texttt{TMP}_1$ `= P * part1` is typed to <`RawBal` , 0, $t_e \times t_{token} \times t_{base}$,-> with no issue. The second part $\texttt{TMP}_2$ `=` $\texttt{TMP}_1$ `+ part2` is problematic by rule R_4 since there is a unit mismatch between $\texttt{TMP}_1$ and `part2`, with the unit $t_{token} \times t_{base}$.

For the example in Figure 2, `signedAmount` and `signedPrice` are typed to <`RawBal` , 0, t_{base}, −> and <`Price` , 0, $t_{quote} \div t_{base}$,-> from previous assignments not shown here. Similarly, `position.base` and `position.quote` are typed to <`RawBal` , 0, t_{base},-> and <`RawBal` , 0, t_{quote},->; `quoteExchange` is typed to <`RawBal` , 0, t_{quote},-> by rule R_6. The code for function `getFee()` is not shown, but its return type (and hence the type of `fee`) is <`Fee` , 0, t_{quote},->. The problematic statement is on line 13, within the true branch. The statement is split into two separate operations (by Slither). The first operation $\texttt{TMP}_1$ `= position.quote-quoteChange` type-checks and results in <`RawBal` , 0, t_{quote},-> by rule R_5. The second operation `newQuote =` $\texttt{TMP}_1$ `+ fee` does not type-check as `Fee` cannot be added to `RawBal` (Table 1 in the supplementary material).

Limitations of Our Type System. Our system is based on the DeFi model in Figure 3, which only abstracts parts of the business models of DeFi projects. It is not quantitative such that our type system cannot detect pure calculation errors. However, our results show that more than half of accounting errors are type errors. In addition, there may be different designs even for the basic operations in Figure 3. For example, interest may not be directly added debt, but rather separately accounted. However, these design choices do not cause problems in our type rules. For instance, we allow interest to be added to debt (Table 1 in supplementary material) but we do not force such addition.

4 EVALUATION

We implement ScType in around 3,000 lines of Python code on Slither [2]. It consists of a type annotation parser, a type propagation system, and a type checking system. It is inter-procedural and cross-contract, meaning that it may automatically include functions from other contracts in analysis (if their code is available). It also handles arrays and object fields. To reduce the overhead of supporting context-sensitivity, it caches analysis results for each function. Details are elided. We aim to address the following research questions.

- **RQ1.** How effective is ScType in disclosing accounting bugs?
- **RQ2.** How efficient is ScType?
- **RQ3.** What are the categories and distributions of accounting bugs?
- **RQ4.** What is the capacity of our type system?
- **RQ5.** How effective is ScType in finding zero-days?

The system and the benchmarks are provided as supplementary material and will be released upon publication.

4.1 Experimental Setup

Benchmark. In the controlled experiments, we utilize the smart contract vulnerabilities collected by Zhang et al. in [1], which details 513 real-world bugs from 113 projects. Of which, 72 were categorized as accounting bugs. We preclude 15 of them due to the inability to be loaded by Slither or missing code. The detailed list of remaining bugs is in Table 3, following their chronical order of being reported. While a project may have multiple contracts, we run ScType on those in which accounting bugs were reported.

Baselines. We ran a few state-of-the-art static analyses on the set of bugs, such as Smartian [14], Slither [2], Oyente [15], and Mythril [16]. However, they could not find these accounting bugs. It is expected as these tools are built for other types of bugs. The results are also consistent with what was reported in [1].

Initial Type Annotations. Although ScType can automatically infer certain type information such as some token units and scaling factors, it may need the user to provide initial information such as financial types for some global variables and some function parameters (if they cannot be inferred). ScType prompts the user for such information when it is missing and cannot be resolved by type inference. Users' efforts are one-time and recorded in a type file for reuse. The information is clear from project description and code comments in most cases, and hence the required user efforts are limited, as demonstrated by the number of annotations in Table 3 (in comparison to the number of functions type-checked). More automation is certainly feasible. For example, financial types can be inferred from variable names in many cases or using mining techniques such as [17, 18]. We leave it to our future work.

The experiments are conducted on a machine with AMD Ryzen 3975x and 512GB RAM.

4.2 RQ1: Effectiveness

The results from running ScType on our benchmark are shown in Table 3. The projects are listed in the leftmost column, followed by a short summary of each project. The number of annotations made is shown in the column starting with A+. The number of functions checked is listed under the Func checked column, which includes functions that are called within other functions. The TW column indicates the number of type warnings by ScType. Singular true positives (TP) and false positives (FP) may both generate multiple warnings (due to the cascading effect of a type error), hence the discrepancy between the warnings amount and the sum of true and false positive amounts. The true positives are listed under the TP column. Accounting bugs that are not type errors (and hence out of scope for ScType) are listed under the NTE, or Not-Type-Error column. Type bugs that are not able to be found with the current system are listed under the MTE, or Missed-Type-Error column.

Observations. In total, we run our tool on 29 projects, covering 57 accounting bugs. ScType reports 29 TPs and 11 FPs. The FPs are mainly due to the path insensitive nature of the tool. We will illustrate with a case later. Even though ScType cannot detect 28 of the 57 bugs, our inspection shows that 24 out of those 28 bugs are not type errors, belonging to other error categories such as pure math errors. Therefore, ScType is able to successfully detect 29/(29+4)=87.9% of accounting type errors. We argue that these results demonstrate the promise of ScType as an attempt to addressing accounting bugs.

False positives. We manually inspect the false positives. We find that the lack of path sensitivity is a major reason. Figure 9 shows an example where ScType fails to type-check `baseAmount` within the function `getRewardShare()`, which takes as parameter an address `token` that represents the address of a currency. `baseAmount` is calculated as an amount of the `token` currency on line 5 via the function `getBaseAmount()`. Then, `token` is compared to the addresses of VADER and USDV tokens by the function calls to `isVADER()` and `isUSDV()` on lines 5 and 9, respectively. If the true branch is taken on line 5, the function `getRewardShare()`

Project Name	Summary	A+	Func checked	TW	FP	TP	NTE	MTE	Analysis Time
MarginSwap	Dex project for margin trading on Uniswap and Sushiswap	17	18	1	0	1	1	0	8.93s
Vader Protocol	Yield project for a collaterized stablecoin	19	53	4	1	2	2	0	28.95s
PoolTogether	Gaming service on yield interest	7	13	1	0	1	1	0	8.94s
Tracer	Derivative project that supports perpetual markets	11	6	1	0	1	1	1	110.57s
Yield Micro	"Lending project supporting borrowing, lending, and liquidity"	4	16	2	0	1	1	1	10.39s
Sushi Trident	Dex project for deploying personalized liquidity markets	24	88	0	0	0	2	0	19.64s
yAxis	Yield project where users' aggregated funds are used in strategies for yield	9	31	4	1	2	1	1	16.81s
Badger Dao	Yield project	9	30	2	0	1	0	0	14.61s
Wild Credit	Lending project relying on pairs of assets instead of a pool	17	108	4	2	1	0	0	45.37s
PoolTogether v4	Gaming service on yield interest	1	16	0	0	0	1	0	14.99s
Sushi Trident p2	Dex project for deploying personalized liquidity markets	24	45	10	4	2	2	0	19.68s
Swivel	Yield project that allows users to set orders for Yield claiming	5	23	2	0	1	0	0	15.03s
Covalent	"Users delegate comissions to a Validators, which stakes the funds for interest"	27	70	1	1	0	0	1	15.87s
Badger Dao p2	Stablecoin to keep the internal contract asset price from fluctuating	8	23	2	0	1	0	0	10.31s
Vader Protocol	Yield project for a collaterized stablecoin (Swap)	34	113	13	2	6	2	0	66.86s
yAxis p2	Yield project where users' aggregated funds are used in strategies for yield	5	8	0	0	0	1	0	9.58s
Malt Finance	"Stablecoin for the contract token, Malt"	27	40	0	0	0	2	0	11.998s
Perennial	Derivative project supporting synthetic token perpetual markets	3	1	1	0	1	0	0	9.35s
Sublime	Lending contract dependent on trust minimization	16	65	5	2	2	0	0	24.54s
Yeti Finance	Lending project made against a contract specific token	10	29	0	0	0	1	0	12.35s
Vader Protocol p3	Yield project for a collaterized stablecoin	4	36	4	0	3	1	0	16.38s
InsureDao	Insurance markets where buyers pay premium for protection against losses	10	61	0	0	0	1	0	14.86s
Rocket Joe	Dex project where users exchange funds in return for new project liquidity	27	37	3	1	1	0	0	2.860s
Concur Finance	Yield project	15	26	0	0	0	1	0	2.980s
Biconomy Hyphen	Cross Chain project where users can deposit and withdraw for pools on different chains	16	61	1	0	1	0	0	14.64s
Sublime	Lending project allowing users to create custom lending pools	8	47	0	0	0	1	0	25.84s
Volt	Dex project which conserves the value of user funds against inflation	1	23	0	0	0	1	0	18.86s
Badger Dao p3	Yield project	8	76	0	0	0	1	0	16.22s
Tigris Trade	Dex project utilizing off-chain oracles to provide real-time prices	33	84	2	0	1	1	0	7.024s
Total				64	14	29	24	4	

* Annotations (A+), Total Warnings (TW), False Positive (FP), True Positive (TP), Not Type Error (NTE), Missed Type Error (MTE)

Table 3: Evaluation Results

```
int usdvReserve; //an amount of USDC
int vaderReserve; //an amount of WETH
function getRewardShare(address token) external {
    uint baseAmount = getBaseAmount(token);
    if (isVader(token)) {
        uint _share = calcShare(_baseAmount,
        usdvReserve, vaderReserve);
        ...
    } else if(isUSDC(token)) {
        uint _share = calcShare(_baseAmount, vaderReserve,
            usdvReserve);
        ...
    }
    ...
}
```

Figure 9: False Positive Example in Vader Protocol p1

swaps `baseAmount` of the VADER token. If not, USDV is swapped. ScType cannot resolve `baseAmount` to a unique token unit. Solving this problem may require path-sensitive analysis such as symbolic execution. We will leave this to our future work.

4.3 RQ2: Efficiency

To answer the research question regarding efficiency, we measure the cost of our tool, which is two-fold: the time of analysis and the total number of annotations. The former is shown in the last column of Table 3. Observe that the tool is very time affordable, with the maximum analysis time being less than 20 seconds. We point out that ScType automatically type-checks all functions that are being called, regardless of whether or not they reside in the same contract. The number of annotations is also reasonable given

```
function belowMaintenanceThreshold(uint256 loan, uinst256
      collateral) external {
  ...
-  return 100 *collateral >= liquidationThresholdPercent *loan;
+  return 100 *collateral < liquidationThresholdPercent *loan;
}
```

Figure 10: Not-Type-Error Example in MarginSwap

the large number of functions checked. Most of the annotations can be derived with minimum one-time manual efforts.

4.4 RQ3: Distribution of Type Errors

We categorize the 33 different accounting type bugs (29 found by ScType and 4 missed) into 3 categories: token unit bugs, scaling factor bugs, and financial type bugs. Their numbers are 10, 12, and 11, respectively, which represent an approximately even distribution. The distribution strongly supports our current design. In addition, we find it a bit counter intuitive that token unit bugs are almost as common as the other two kinds, although they are simpler.

4.5 RQ4: Capacity of Type System

As shown in Table 3, ScType cannot detect 28 accounting bugs. Out of the 28, 24 are not type errors and deemed out-of-scope, while 4 are type errors that currently cannot be handled. In this section, we provide two case studies to illustrate these two types of bugs. Figure 10 shows a typical not-type-error (NTE). The bug lies in that the developers used the wrong comparison. Other NTE types include coefficient errors and even use of wrong formulas. These

```
uint256 totalReserve;
uint256 strategyReserve;
function setCap(uint256 cap) external {
    ...
    diff = strategyReserve - cap
-   totalReserve -= strategyReserve;
+   totalReserve -= diff;
    ...
}
```

Figure 11: Missed-Type-Error Example in yAxis p1

bugs need stronger oracles than type rules, such as input-output pairs and formal specification of business models. Figure 11 shows a missed-type-error (MTE). This function computes the amount that a certain reserve of some *strategy* (e.g., a contract that yields) exceeds a cap and removes the excess from the total reserve. In particular, the specific reserve is denoted by variable `strategyReserve`, the cap is `cap`, the excess is `diff` and the total reserve is `totalReserve`. The bug lies in that developers incorrectly subtract `strategyReserve`, while they should subtract `diff`. Although it is beyond our current system, a stronger type system that models balance delta such as `diff` as well as balance upper-bound like `cap` may prevent this bug. We leave this to our future work.

4.6 RQ5: Finding Zero-days

To study the real-world impact of ScType, we use it to audit a large real-world contract through Code4Rena. The project has over 10 thousands lines of code. We applied the technique on 9 contracts, found and reported 6 zero-days, with 4 of them leading to direct fund loss. Three of them are financial type errors and the other three are token unit bugs. We have created exploit inputs for these bugs as proof-of-concepts. According to Code4Rena's policy, details should not be made public until the judges inspect all the bug reports and the developers are given the chances to fix the bugs.

5 THREATS TO VALIDITY

There is *internal* threat to validity due to human mistakes in type annotations. In practice, these annotations are mostly obvious from documentation and variable names. For example, fee and debt variables tend to have subwords "`fee`" and "`debt`" in their variable names. In addition, there may be implementation errors. As wrong annotations/implementations lead to spurious type errors, the few false positives by ScType indicate that the threat is mitigated. In the future, we plan to further reduce the human efforts (and hence the internal threat) by mining variables' financial meanings. The *external* threat mainly lies in the subjects used in our study. We mainly use the bugs in [1], which may not be representative. The risk is mitigated as all projects in the benchmark are real-world applications, with many having high complexity. The bug reports had gone through multiple rounds of interactions between auditters, developers, and Code4rena judges. In addition, we recently apply ScType to a very complex project and have encouraging results.

6 RELATED WORK

Detecting Business-related Vulnerabilities. Accounting bugs are related to business models. There have been pioneering efforts in detecting business related bugs. Wang et al. [19] proposed a fuzzing tool *Vultron*, which developed an interesting observation regarding *balance* and *transaction invariants*. In particular, the total balance of all the users and the contract should be the same, and transactions in or out of the contract should correspond to the same increase or decrease in total balance. This prevents bugs such as not updating a contract's balance after a withdraw. They later developed mining techniques to infer these invariants [17, 18, 20]. Fairness bug detection [21] aimed to detect unfair behaviors for game-like contracts having multiple participants. Sun et al. [22] developed a method to detect smart contract vulnerabilities based on a swap invariant and a transfer invariant, or logical rules that must be followed in order to facilitate proper functions. The technique can detect overflows and unprotected asset increases. In comparison, ScType can detect type problems that are largely complementary to the above works. Verification techniques [23–30] are capable of detecting a wide spectrum of bugs including accounting bugs if the user can provide the specifications. In comparison, ScType encodes properties in its type rules.

Abstract Type Inference and Checking. ScType is essentially an abstract type system [31–40] that derives abstract types with much richer semantics than primitive types. However, existing techniques do not focus on smart contracts, which have very unique finance oriented semantics. Tan et al. [30] developed a refinement type system known as *SolType* for Solidity. It models low level relationships between integers and checks for overflows/underflows.

Smart Contract Bug Finding. ScType is related to smart contract bug finding in general, including static analysis [2, 41–50], fuzzing [14, 19, 51–60], and symbolic execution [15, 37, 55, 61–67]. In contract, ScType focuses finding accounting bugs, complementary to these techniques.

Bug Studies. We are inspired by a recent study on smart contract vulnerabilities [1], which showed the prevalence of accounting bugs and the difficulty of finding them, and also by a list of other comprehensive studies of various kinds of smart contract bugs and programming practices [68–75].

7 CONCLUSION

We develop an abstract type inference and checking technique to detect accounting bugs in smart contracts, a kind of bug difficult for existing automatic tools. The technique models token units, scaling factors, and financial meanings of individual variables and checks type consistencies. Our results show that more 58% of known accounting bugs are type errors, and our tool detects 87.9% of these type errors. It also finds 6 zero-days.

REFERENCES

[1] Z. Zhang, B. Zhang, W. Xu, and Z. Lin, "Demystifying exploitable bugs in smart contracts," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 615–627, 2023.

[2] J. Feist, G. Grieco, and A. Groce, "Slither: a static analysis framework for smart contracts," in *WETSEB@ICSE*. IEEE / ACM, 2019.

[3] "Code4rena." [Online]. Available: https://code4rena.com

[4] "Sctype," 2023. [Online]. Available: https://github.com/NioTheFirst/ScType/tree/ICSE24Artifact

[5] "Vader protocol p1 project," 2023. [Online]. Available: https://github.com/ZhangZhuoSJTU/Web3Bugs/tree/main/contracts/5/vader-protocol

[6] "Tracer project," 2023. [Online]. Available: https://github.com/ZhangZhuoSJTU/Web3Bugs/tree/main/contracts/16

[7] "Perpetual markets," 2023. [Online]. Available: https://milkroad.com/funding/perpetual-contracts/

[8] "Funding fee," 2023. [Online]. Available: https://www.binance.com/en/blog/futures/what-are-funding-fees-in-binance-futures-6595842576313788144

[9] "Tigris trade project," 2023. [Online]. Available: https://github.com/ZhangZhuoSJTU/Web3Bugs/tree/main/contracts/192

[10] "Biconomy project," 2023. [Online]. Available: https://github.com/ZhangZhuoSJTU/Web3Bugs/tree/main/contracts/97

[11] R. F. Muth, "The derived demand curve for a productive factor and the industry supply curve," *Oxford Economic Papers*, vol. 16, no. 2, 1964.

[12] "Defillama." [Online]. Available: https://defillama.com/

[13] "defillama/categories," 2023. [Online]. Available: https://defillama.com/categories

[14] J. Choi, D. Kim, S. Kim, G. Grieco, A. Groce, and S. K. Cha, "Smartian: Enhancing smart contract fuzzing with static and dynamic data-flow analyses," in *ASE*. IEEE, 2021.

[15] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016.

[16] "Consensys/mythril," 2022. [Online]. Available: https://github.com/ConsenSys/mythril

[17] Y. Liu, Y. Li, S.-W. Lin, and C. Artho, "Finding permission bugs in smart contracts with role mining," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[18] Y. Liu, "A unified specification mining framework for smart contracts," *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[19] H. Wang, Y. Li, S.-W. Lin, L. Ma, and Y. Liu, "Vultron: catching vulnerable smart contracts once and for all," in *ICSE-NIER*. IEEE, 2019.

[20] Y. Liu and Y. Li, "Invcon: A dynamic invariant detector for ethereum smart contracts," *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[21] Y. Liu, Y. Li, S.-W. Lin, and R.-R. Zhao, "Towards automated verification of smart contract fairness," *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020.

[22] J. lei Sun, S. Huang, X. Wang, M. Wang, and J. Du, "A detection method for scarcity defect of blockchain digital asset based on invariant analysis," *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, pp. 73–84, 2022.

[23] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar, "Online detection of effectively callback free objects with applications to smart contracts," *Proc. ACM Program. Lang.*, vol. 2, no. POPL, pp. 48:1–48:28, 2018.

[24] Á. Hajdu and D. Jovanovic, "solc-verify: A modular verifier for solidity smart contracts," in *VSTTE*, ser. Lecture Notes in Computer Science, vol. 12031. Springer, 2019, pp. 161–179.

[25] Y. Wang, S. K. Lahiri, S. Chen, R. Pan, I. Dillig, C. Born, I. Naseer, and K. Ferles, "Formal verification of workflow policies for smart contracts in azure blockchain," in *VSTTE*, ser. Lecture Notes in Computer Science, vol. 12031. Springer, 2019, pp. 87–106.

[26] S. So, M. Lee, J. Park, H. Lee, and H. Oh, "VERISMART: A highly precise safety verifier for ethereum smart contracts," in *IEEE Symposium on Security and Privacy*. IEEE, 2020, pp. 1678–1694.

[27] J. Jiao, S. Kan, S.-W. Lin, D. Sanán, Y. Liu, and J. Sun, "Semantic understanding of smart contracts: Executable operational semantics of solidity," *2020 IEEE Symposium on Security and Privacy (SP)*, pp. 1695–1712, 2020.

[28] P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li, "A survey of smart contract formal specification and verification," *ACM Computing Surveys (CSUR)*, vol. 54, pp. 1 – 38, 2020.

[29] J. Jiao, S.-W. Lin, and J. Sun, "A generalized formal semantic framework for smart contracts," *Fundamental Approaches to Software Engineering*, vol. 12076, pp. 75 – 96, 2020.

[30] B. Tan, B. Mariano, S. K. Lahiri, I. Dillig, and Y. Feng, "Soltype: refinement types for arithmetic overflow in solidity," *Proceedings of the ACM on Programming Languages*, vol. 6, no. POPL, pp. 1–29, 2022.

[31] M. Karr and D. B. Loveman, "Incorporation of units into programming languages," *Commun. ACM*, vol. 21, pp. 385–391, 1978.

[32] S. Hangal and M. S. Lam, "Automatic dimension inference and checking for object-oriented programs," *2009 IEEE 31st International Conference on Software Engineering*, pp. 155–165, 2009.

[33] P. J. Guo, J. H. Perkins, S. McCamant, and M. D. Ernst, "Dynamic inference of abstract types," in *International Symposium on Software Testing and Analysis*, 2006.

[34] V. Raychev, M. T. Vechev, and A. Krause, "Predicting program properties from "big code"," *ACM SIGPLAN Notices*, vol. 50, pp. 111 – 124, 2015.

[35] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: specification inference for explicit information flow problems," in *ACM-SIGPLAN Symposium on Programming Language Design and Implementation*, 2009.

[36] S. Kate, J.-P. Ore, X. Zhang, S. Elbaum, and Z. Xu, "Phys: Probabilistic physical unit assignment and inconsistency detection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, 2018, p. 563–573.

[37] S. So, S. Hong, and H. Oh, "SmarTest: Effectively hunting vulnerable transaction sequences in smart contracts through language Model-Guided symbolic execution," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021.

[38] A. J. Kennedy, "Dimension types," in *European Symposium on Programming*, 1994.

[39] M. Allamanis, E. T. Barr, S. Ducousso, and Z. Gao, "Typilus: neural type hints," *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[40] J.-P. Ore, C. Detweiler, and S. G. Elbaum, "Lightweight detection of physical unit inconsistencies without program annotations," *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017.

[41] T. Chen, X. Li, X. Luo, and X. Zhang, "Under-optimized smart contracts devour your money," in *SANER*. IEEE Computer Society, 2017, pp. 442–446.

[42] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev, "Securify: Practical security analysis of smart contracts," in *CCS*. ACM, 2018, pp. 67–82.

[43] L. Brent, A. Jurisevic, M. Kong, E. Liu, F. Gauthier, V. Gramoli, R. Holz, and B. Scholz, "Vandal: A scalable security analysis framework for smart contracts," *arXiv preprint arXiv:1809.03981*, 2018.

[44] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis, "Madmax: Surviving out-of-gas conditions in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 2, no. OOPSLA, 2018.

[45] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita, and H. Kurihara, "Security assurance for smart contract," in *NTMS*. IEEE, 2018.

[46] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov, "Smartcheck: Static analysis of ethereum smart contracts," in *WETSEB@ICSE*. ACM, 2018, pp. 9–16.

[47] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "Zeus: analyzing safety of smart contracts." in *NDSS*, 2018.

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

[49] S. Wang, C. Zhang, and Z. Su, "Detecting nondeterministic payment bugs in ethereum smart contracts," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, 2019.

[50] J. Huang, K. Zhou, A. Xiong, and D. Li, "Smart contract vulnerability detection model based on multi-task learning," *Sensors*, 2022.

[51] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe, "Reguard: finding reentrancy bugs in smart contracts," in *ICSE-Companion*. IEEE, 2018.

[52] B. Jiang, Y. Liu, and W. K. Chan, "Contractfuzzer: Fuzzing smart contracts for vulnerability detection," in *ASE*. IEEE, 2018.

[53] T. D. Nguyen, L. H. Pham, J. Sun, Y. Lin, and Q. T. Minh, "sfuzz: an efficient adaptive fuzzer for solidity smart contracts," in *ICSE*. ACM, 2020, pp. 778–788.

[54] H. Wang, Y. Liu, Y. Li, S.-W. Lin, C. Artho, L. Ma, and Y. Liu, "Oracle-supported dynamic exploit generation for smart contracts," *IEEE Transactions on Dependable and Secure Computing*, 2020.

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

[56] A. Groce and G. Grieco, "echidna-parade: a tool for diverse multicore smart contract fuzzing," in *ISSTA*. ACM, 2021, pp. 658–661.

[57] G. Grieco, W. Song, A. Cygan, J. Feist, and A. Groce, "Echidna: effective, usable, and fast fuzzing for smart contracts," in *ISSTA*. ACM, 2020, pp. 557–560.

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

[59] J. He, M. Balunovic, N. Ambroladze, P. Tsankov, and M. T. Vechev, "Learning to fuzz from symbolic execution with application to smart contracts," in *CCS*. ACM, 2019, pp. 531–548.

[60] V. Wüstholz and M. Christakis, "Harvey: a greybox fuzzer for smart contracts," in *ESEC/SIGSOFT FSE*. ACM, 2020, pp. 1398–1409.

[61] I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor, "Finding the greedy, prodigal, and suicidal contracts at scale," in *ACSAC*. ACM, 2018, pp. 653–663.

[62] J. Krupp and C. Rossow, "teether: Gnawing at ethereum to automatically exploit smart contracts," in *USENIX Security Symposium*. USENIX Association, 2018, pp. 1317–1333.

[63] M. Mossberg, F. Manzano, E. Hennenfent, A. Groce, G. Grieco, J. Feist, T. Brunson, and A. Dinaburg, "Manticore: A user-friendly symbolic execution framework for binaries and smart contracts," in *ASE*. IEEE, 2019.

[64] J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang, "scompile: Critical path identification and analysis for smart contracts," in *ICFEM*, ser. Lecture Notes in Computer Science, vol. 11852. Springer, 2019, pp. 286–304.

[65] Z. Wang, B. Wen, Z. Luo, and S. Liu, "Mar: A dynamic symbol execution detection method for smart contract reentry vulnerability," in *International Conference on Blockchain and Trustworthy Systems*. Springer, 2021.

[66] "Blockchain technology solutions." [Online]. Available: https://consensys.net/

[67] P. Bose, D. Das, Y. Chen, Y. Feng, C. Kruegel, and G. Vigna, "Sailfish: Vetting smart contract state-inconsistency bugs in seconds," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[68] J. Chen, X. Xia, D. Lo, J. C. Grundy, X. Luo, and T. Chen, "Defining smart contract defects on ethereum," *IEEE Trans. Software Eng.*, vol. 48, no. 2, pp. 327–345, 2022.

[69] P. Zhang, F. Xiao, and X. Luo, "A framework and dataset for bugs in ethereum smart contracts," in *ICSME*. IEEE, 2020, pp. 139–150.

[70] W. Dingman, A. Cohen, N. Ferrara, A. Lynch, P. Jasinski, P. E. Black, and L. Deng, "Classification of smart contract bugs using the NIST bugs framework," in *SERA*. IEEE, 2019, pp. 116–123.

[71] D. Perez and B. Livshits, "Smart contract vulnerabilities: Vulnerable does not imply exploited." in *USENIX Security Symposium*, 2021.

[72] W. Zhang, L. Wei, S. C. Cheung, Y. Liu, S. Li, L. Liu, and M. R. Lyu, "Combatting front-running in smart contracts: Attack mining, benchmark construction and vulnerability detector evaluation," *IEEE Transactions on Software Engineering*, vol. 49, pp. 3630–3646, 2022.

[73] V. Piantadosi, G. Rosa, D. Placella, S. Scalabrino, and R. Oliveto, "Detecting functional and security-related issues in smart contracts: A systematic literature review," *Software: Practice and Experience*, vol. 53, pp. 465 – 495, 2022.

[74] J. Chen, X. Xia, D. Lo, and J. C. Grundy, "Why do smart contracts self-destruct? investigating the selfdestruct function on ethereum," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, pp. 1 – 37, 2020.

[75] J. Chen, X. Xia, D. Lo, J. C. Grundy, and X. Yang, "Maintenance-related concerns for post-deployed ethereum smart contract development: issues, techniques, and future challenges," *Empirical Software Engineering*, vol. 26, 2021.