SmartInv: Multimodal Learning for Smart Contract Invariant Inference

Sally Junsong Wang*, Kexin Pei*†, Junfeng Yang*
*Columbia University, NY †The University of Chicago, IL
jw4074@columbia.edu, kpei@cs.uchicago.edu, junfeng@cs.columbia.edu

2024 IEEE Symposium on Security and Privacy (SP) | 979-8-3503-3130-1/24/$31.00 ©2024 IEEE | DOI: 10.1109/SP54263.2024.00126

***Abstract*—Smart contracts are software programs that enable diverse business activities on the blockchain. Recent research has identified new classes of "machine un-auditable" bugs that arise from source code not meeting underlying transaction contexts. Existing detection methods require human understanding of underlying transaction logic and manual reasoning across different sources of context (i.e., modalities), such as code and natural language specifying the expected transaction behavior.**

To automate the detection of "machine un-auditable" bugs, we present SmartInv, an accurate and fast smart contract invariant inference framework. Our key insight is that the expected behavior of smart contracts, as specified by invariants, relies on understanding and reasoning across *multimodal* information, such as source code and natural language. We propose a new finetuning and prompting strategy to foundation models, Tier of Thought (ToT), to reason across multiple modalities of smart contracts and to generate invariants. SmartInv then localizes potential vulnerabilities by checking the violation of those generated invariants.

We evaluate SmartInv on real-world smart contract bugs that resulted in financial losses over the past 2.5 years (from January 1, 2021 to May 31, 2023). Extensive evaluation shows that SmartInv can generate useful invariants to effectively localize "machine un-auditable" bugs, from which SmartInv uncovers 119 zero-day bugs. We sampled eight bugs and reported them to the respective developers. Six vulnerabilities were quickly fixed by the developers, five of which are confirmed as "high severity."

1. Introduction

Bugs in smart contracts are often serious vulnerabilities that lead to significant loss of funds. In 2022 alone, $2 billion was lost due to smart contract bugs [62], [92]. What makes smart contract bugs particularly damaging is the fact that once a smart contract is deployed, it becomes immutable, making it difficult to fix any vulnerabilities in the code.

Recent research has identified a new category of bugs known as "machine un-auditable" *functional* bugs, which are prevalent in smart contracts (Table 1). These bugs, as their name suggests, cannot be reliably detected using existing automated tools that rely on pre-defined bug patterns [92]. Unlike implementation bugs (e.g., integer overflows), which often exhibit universal patterns that can be easily checked, functional bugs arise from a failure to reason about extensive domain-specific properties, e.g., a specific transaction context. Detecting functional bugs requires nontrivial reasoning across multiple sources of information or *modalities*, such as source code and transaction logic additionally implied in natural language documentation.

TABLE 1: Statistics on bountied vulnerabilities of Solidity-based smart contracts (from September, 2021 to May, 2023)

Implementation	Functional	Others	Total
929 (17.52%)	4,305 (81.20%)	68 (1.28%)	5,302

Existing smart contract bug detection tools rely heavily on human experts to express their knowledge of transaction context into bug specifications [7], [32], [70], [87], and are thus often tailored for specific bug types and do not scale to a large number of programs. While some approaches automate the construction of bug specifications [88], they are limited to those leaving explicit patterns in the transaction history and do not generalize to most functional bugs that would otherwise require a deep understanding of the expected transaction context.

Listing 1 shows an example of a functional bug. The `getPrice()` function computes `price` by the ratio of `token0` and `token1` balances in `address(this)`. Ideally, `price` is expected to remain stable within a range. However, as `token0` and `token1` are state variables, i.e., static variables, they can be easily manipulated by external parties. Therefore, when `getPrice()` is invoked to return `price`, its return value can fluctuate significantly, leading to the potential exploit of unexpected arbitrage on top of the newly manipulated price difference. The functional bug of `getPrice()` stems from not effectively meeting transaction requirements on the intended range of price. For example, inflating `token1`'s balance via another malicious contract can throw `price` off its intended range (see §2.2 for details). Such a functional bug cannot be detected without first understanding how `getPrice()` would participate in the transactions.

We present SmartInv, an automated framework based on foundation models to infer smart contract invariants and to detect bugs at scale. While there are machine learning approaches for generating invariants [31], [46], [64], [86], [89], they predate foundation models, and thus follow the typical paradigm in hand-engineering limited features that can be helpful for contract invariant inference. The unique

```
uint price;
IERC20 token0, token1;

function getPrice() {
  //functional bug: price manipulation
  price = token1.balanceOf(address(this))/
      token0.balanceOf(address(this));
}
```

Listing 1: Functional bug example. An attacker can pump up `price` by inflating `token1`'s balance.

feature of SMARTINV, which differentiates from existing analyzers, is that SMARTINV leverages foundation models to reason about multimodal inputs, such as source code and natural language (comments, meaningful function signatures, and documentation describing transaction scenarios). Foundation models are particularly suited for analyzing multimodal information and domain-specific bug detection, because they are pretrained on both natural language texts and code, and can be further finetuned for domain-specific knowledge.

To reason across multiple modalities, we develop *Tier of Thought (ToT)*, a new prompting strategy, that can be used to finetune and elicit explicit reasoning of foundation models on the program structure of smart contracts. In contrast to other foundation-model-based approaches [13], [16], [72], ToT applies universally across contract types, eliminating the need for bug-specific reasoning heuristics. ToT breaks down the process of invariant generation into intermediate abstract tiers based on the typical reasoning steps that human analyzers would take, such as predicting critical program points to check invariants, generating invariants associated with the predicted program points, and ranking the invariants by predicting their likelihood of preventing bugs. Based on the ranked invariants, SMARTINV can efficiently verify the invariants by prioritizing the invariants that are more likely bug-preventive using a bounded model checker without exhaustively enumerating all of them.

Contributions. We make the following contributions:

- To the best of our knowledge, SMARTINV proposes the first finetuning approach that can both infer invariants and detect bugs by reasoning across multiple smart contract modalities, critical to detecting functional bugs pre-deployment.
- We design a new prompting approach to foundation models, Tier of Thought (ToT), to finetune and elicit their reasoning by following the thought process of human analyzers, significantly improving the accuracy of the generated invariants and detected bugs while reducing runtime overhead.
- We unite the potential of finetuning foundation models with the soundness of formal verification to mitigate hallucination. One key benefit of inferring invariants using foundation models is that the inferred invariants can be validated by verification tools.
- Our extensive evaluation demonstrates that SMARTINV is effective in invariant generation and functional bug localization. Notably, SMARTINV has found 119 zero-day bugs in the wild. We sampled eight bugs and reported them to the respective developers. Six vulnerabilities were quickly fixed by the developers, five of which are confirmed as "high severity."
- We collect a large (2,173 samples) annotated smart contract invariant dataset for training and 89,621 real-world contracts for bug detection. We release the datasets and the tool for public use at https://github.com/columbia/SmartInv.

TABLE 2: SMARTINV detected bug classification. Modalities: the minimum modalities required to detect a given bug class. SC: source code. NL: natural language. Pattern Detectable: a bug class can be detected by identifying low-level coding patterns. ToT Detectable: a bug class can be detected by Tier of Thought. ◐: a modality is sometimes required.

	Modalities		Pattern	ToT
	SC	NL	Detectable	Detectable
Implementation Bugs				
Reentrancy (RE) [44]	✓	✗	✓	✓
Integer overflow/underflow (IF) [12]	✓	✗	✓	✓
Arithmetic flaw (AF) [67]	✓	✗	✓	✓
Suicidal contract (SC) [54]	✓	✗	✓	✓
Ether leakage (EL) [54]	✓	✗	✓	✓
Insufficient gas (IG) [51]	✓	✗	✓	✓
Incorrect visibility/owner (IVO) [28]	✓	✗	✓	✓
Functional Bugs				
Price manipulation (PM) [84]	✓	✓	✗	✓
Privilege escalation (PE) [11]	✓	✓	✗	✓
Atomicity violation (AV) [66]	✓	✓	✗	✓
Business logic flaw (BLF) [27]	✓	◐	✗	✓
Inconsistent state update (IS) [92]	✓	✓	✗	✓
Cross bridge (CB) [88]	✓	◐	✗	✓
ID uniqueness violation (IDV) [92]	✓	✓	✗	✓

2. Overview

This section introduces the necessary background (§2.1), two motivating examples (§2.2), followed by an overview of SMARTINV's workflow (§2.3).

2.1. Background

Smart Contract Modalities. Smart contract modalities can be broadly understood as sources of information under which bug-preventive invariants are generated. Accordingly, smart contracts contain two main modalities: i) contract source code; ii) natural language, usually in the form of implementation-related comments and domain-specific text description, e.g., documentation. From the contracts we studied in Table 1, 5,265 contracts (99.31%) contain natural language related to code logic and expected transaction behavior. Existing invariants generators [9] and bug detectors [10], [69], [70] focus only on a single modality, namely contract source code. To our knowledge, SMARTINV is the first smart contract analysis tool that can reason across both modalities.

Bug Taxonomy. As Table 2 demonstrates, SmartInv-detected bugs can be categorized into two types:

- *Implementation bugs*: these bugs are characterized by certain patterns in source code, such as reentrancy (RE), which has a representative pattern of cyclic transactions. Reasoning about implementation bugs does not require domain-specific properties or multimodal information.
- *Functional bugs*: these bugs are tied to highly specialized transaction contexts and domain-specific properties. Detecting these bugs requires understanding and reasoning across multimodal contract information. Functional bugs are usually not pattern detectable from source code.

Implementation bugs usually do not require domain-specific information, so they can be detected based on general patterns [15], [18], [22], [26], [52], [53], [69], [70], [75], [76], [79], [85] without relying on any multimodal hints. For example, integer over/underflows (IF) can be detected by general test suites similar to buffer overflow [41] used in traditional software. Reentrancy (RE) can be detected by testing general cyclic transaction patterns.

Functional bugs arise from highly domain-specific transaction contexts and exhibit unintended behavior under dynamic transaction executions, i.e., incorrect stateful transitions and/or inter-contract communications given specific domains. Detecting functional bugs requires an expert understanding of domain-specific properties and transaction contexts. For example, a smart contract may contain some formula to calculate price, and a comment describing the transaction context is that "price should stay within a certain range based on market trends from day 1 to day 30." These transaction contexts cannot be easily captured by code and are often overlooked by developers.

Smart Contract Invariants. Invariants specify smart contract properties as logic predicates that should hold true at specified program points. For example, the invariants that check for implementation bugs, e.g., integer overflow, can be simply written as `assert(x<uint256(-1))`. The invariants that check for functional bugs by enforcing a specific balance constraint can be written as `assert(1<balance<1321)`. We have built a range of invariant types into SmartInv (see §3.2 for details).

Prompting Foundation Models. Foundation models (or Large Language Models) are pre-trained on texts and code with a large number of parameters [37]. They are known for their "pre-training, finetuning, and prompting" paradigm for training and inference [47]. In pre-training and finetuning, the model is trained to estimate probabilistic distribution over a sequence of tokens and generate new tokens based on prior tokens in the input token sequence. In prompting, the trained model takes as input a sequence of tokens, and outputs the next tokens iteratively until reaching the maximum token length or end of the sequence, i.e., a period. For example, given an invariant generation task, a prompt can be a question, such as "[smart contract code snippet], what are the invariants at line 1?" Recent research [82], [83] has constructed prompts in different formats such that these prompts elicit intermediate reasoning steps of foundation models. Our prompts (described in §3.2) elicit such intermediate reasoning steps.

2.2. Motivating Examples

We provide two real-world hacks (simplified for readability) as motivating examples in this section. We refer to the line number of a code statement as a *program point* and the line number where invariants should be inserted as a *critical program point*.

Example 1: Flashloan Primer. Flashloans in smart contracts are uncollateralized and allow users to borrow assets without any cost as long as users pay back *within a single transaction*. Tokens declared with external libraries and asset-swapping contracts that support external calls are thus at risk to flashloans. Listing 2 shows a real-world example simplified to demonstrate how malicious users take flashloans to exploit Visor [19], [21], [29], a money market contract providing liquidity services. Users can deposit tokens by calling the `deposit` function at line 15 to mint shares of `poolToken` in return. A malicious user, however, can borrow flashloans to inflate `price` and to mint an outsized share of `poolToken`.

Specifically, an attacker can take a flashloan of `token1` and swap `token1` for `token0`. Given the automated market maker nature of the contract, the swap will spike up the real-time or spot price of `token0` in `token1`. The attacker invokes the `deposit` function to deposit `token0` into the contract. The contract then calculates an outsized share of `poolToken` to transfer to the attacker based on the manipulated `price`. Afterwards, the attacker can sell the `poolToken` in another market to repay the flashloaned `token1`. For interested readers, full mathematical details and developer's solution to the bug can be found at [3], [4].

Existing prompting frameworks [13], [16], [72] specify no invariants and identify incorrect bugs such as reentrancy. Existing bug analyzers based on formal verification, symbolic execution, and other dynamic analysis [48], [69], [69], [70] report Listing 2 as a healthy contract, because they analyze only source code without considering the domain-specific price oracle context implied by the blue comments.

However, analyzing the Listing 2 bug requires understanding the natural language hints indicating that the price oracle is vulnerable to real-time price volatility. Table 3 highlights SmartInv's solution using invariants (discussion in §3) by reasoning across source code and natural language hints. SmartInv infers the lines immediately after lines 17 as a critical program point, and infers an invariant `assert(price <= Old(price)*k)`. Similar to the use of `Orig()` in Daikon [17] and `Old()` ESCJML [56], `Old(price)` returns the previous price point the last time the `deposit()` function is invoked. k has a default value of 2 in SmartInv and can be updated based on developers' desired volatility ratio. Any violation of the assertion invariant would signal price volatility exceeding user's desired k and thus would signal price manipulation.

```
1  contract simplifiedVisor{
2      /*pool token issued by this contract*/
3      IERC20 poolToken;
4      /*two types of token reserves*/
5      IERC20 token0, token1;
6      /*reporting price at real time*/
7      uint price;
8
9      /* real-time price updates by the ratio of token
          reserves*/
10     function getRealPrice() internal {
11         //SmartInv: possible flashloan
              injection
12         price = token1.balanceOf(address(this))
              /token0.balanceOf(address(this));
13     }
14     //SmartInv: minting shares by deposits
15     function deposit(uint deposit0, address to)
           public {
16         /* price may change */
17         getRealPrice();
18         uint adjust = price*deposit0;
19         unit shares = adjust * ... // calculate
              the shares of pool tokens to mint
20         _mint(to, shares);
21     }
22 }
```

Listing 2: Functional buggy snippet from the spot price manipulation of Visor [5], [21] (simplified for readability).

Note that even without violating the invariant, an attacker may still be able to launch a flashloan attack, and the developer's fix for this issue is to use the time-weighted average price instead of the spot price. The key benefit of SMARTINV-inferred invariant in this example is that it pinpoints the potential price manipulation vulnerability for developer fix.

Example 2: Voting Fraud. The voting fraud bug [24], [92] in Listing 3 is officially recognized by the National Common Vulnerabilities and Exposures (CVE) with an assigned ID [58]. This hack was made possible by flashloans and classified as privilege escalation under functional bug types. The contract developers were aware of the potential for flashloan attacks, so they tried to mitigate the risk by restricting the order in which the `startExecute()`, `execute()`, and `endExecute()` functions could be invoked.

If a proposal is not ongoing and sTime = 0, then a message sender can invoke `startExecute()`. If a proposal is ongoing (sTime != 0 and sTime + 24 hours $>$ block.timestamp), then a message sender can only invoke `execute()`. Otherwise (after 24 hours has passed), the proposal round can be ended by invoking `endExecute()`. As the contract developer(s) intended, `startExecute()` must be invoked before `execute()` within a proposal. `execute()` and `endExecute()` cannot be invoked within a single transaction (or within a 24-hour proposal round) to prevent flashloan attacks. However, the key vulnerability lies in the developers' assumption that the three functions, `startExecute()`, `execute()`, and `endExecute()`, would be invoked sequentially in a proposal round. Unfortunately, that assumption does not hold. An attacker can bypass the `execute()` function by invoking the `endExecute()` directly after taking a flashloan to become the highest proposer.

TABLE 3: SMARTINV inferred invariants in Listing 2. 17+ refers to the line-numbered location where the invariant should be inserted, e.g., 17+ means immediately after line 17. `Old(price)` evaluates a variable's pre-state and returns the price point before `liquidate` function is called. k is an adjustable ratio, where SMARTINV sets default as k=2.

Critical Program Points	Inferred Invariants
17+	`assert(price <= Old(price)*k);`

The attack above is possible because the `votingToken` variable is declared with the IERC20 wraparound library contract, which tracks how many tokens a user would like to vote when calling the `execute()` function. The IERC20 wraparound library contract also has its own `transferForm()` function. As a result, any variable declared with IERC20 can invoke `transferForm()` directly.

Suppose a hacker borrows a large flashloan and injects it into `votingToken` via the `transferForm()` function to make the highest bid right after the 24-hour voting window expires (so the flashloan can be paid back within a single transaction). This bypasses the `execute()` function and allows the hacker to invoke `transferForm()` directly in the IERC20 library contract. Immediately after the voting window expires, the hacker ends the proposal round by invoking the `endExecute()` function. This exploit allows the hacker to become the new contract owner at line 27 and thus to invoke the highly privileged `getFunds()` function at line 32. The attacker can then obtain all locked tokens and pay back the flashloan with a profit within the same transaction.

Existing analyzers [52], [69], [70] mistakenly report that Listing 3 contract contains an integer overflow/underflow bug related to $sTime$ at lines 23 and 24 (false positives because Solididy version $\geqslant$ 0.8 automatically preempts operations causing integer overflow/underflow) while omitting the more damaging privilege escalation bug. Their mistaken reporting stems from relying on pattern-matching arithmetic operations without considering the underlying transaction logic.

SMARTINV's solution is to reason across source code and natural language hints in blue (comments and variable names related to the transaction context). First, from the source code and comments, SMARTINV infers that the transaction context is "bidding." After predicting "bidding" transaction context, SMARTINV infers critical program points and invariants as highlighted in Table 4. If a malicious actor bypasses the `execute()` function and injects a large flashloan in `endExecute()` function directly, then `votingToken` variable would transition to a wrong state that violates the assertion invariant (`Old(votingToken.balanceOf(address(this))` `==votingToken.balanceOf(address(this))`) after line 25.

```
1  contract TimelockController {
2    /*this is a bidding contract:
3    watch out for flashloan */
4    struct Proposal {
5      uint sTime; address newOwner;
6    }
7    IERC20 votingToken; /*important variable*/
8    address owner;
9    Proposal proposal;
10
11   /*the following three functions should be
         executed atomically */
12   function startExecute() external {
13     require(proposal.sTime == 0, "on-going proposal");
14     proposal = Proposal(block.timestamp, msg.sender);
15   }
16
17   function execute(uint amount) external {
18     require(proposal.sTime + 24 hours > block.
           timestamp, "execution has ended");
19     votingToken.transferFrom(msg.sender, address(this)
           , amount);
20   }
21
22   function endExecute() external {
23     require(proposal.sTime != 0, "no proposal");
24     require(proposal.sTime + 24 hours < block.
           timestamp, "execution has not ended");
25     require(votingToken.balanceOf(address(this)) * 2 >
            votingToken.totalSupply(),"execution failed"
           );
26     /*we're about to change the owner of the contract */
27     owner = findHighest(_allProposals);
28     delete proposal;
29   }
30
31   /*highest proposer becomes the new owner of the
         contract and gets all locked funds*/
32   function getFunds() external onlyOwner {
33     ...
34     return allLockedTokens;
35   }
36 }
```

Listing 3: Openzepplin vulnerability (reported in CVE-2021-39168 and simplified for readability).

2.3. SmartInv Workflow

Figure 1 shows SmartInv's workflow. SmartInv first finetunes the model on a dataset of labeled contracts with Tier of Thought (ToT) prompts (①). SmartInv learns to minimize cross-entropy loss [91] between ground truth and inferred answers (②). During inference, SmartInv takes a previously unseen new contract as input and prompts the finetuned model using ToT (③). We develop a new iterative prompting process: SmartInv uses the answers from prior easier tiers to guide answer generation for subsequent more challenging tiers. After the finetuned model generates invariants (④), SmartInv proceeds to verify inferred invariants by proving program correctness at (⑤). If no proof of program correctness is found after the initial verification, SmartInv uses a bounded model checker to seek violations (counterexamples) of inferred invariants (⑥). Finally, SmartInv outputs a report on verified invariants and detected bugs. Once finetuned, SmartInv is fully automated to infer invariants and detect bugs.

In building this workflow, there are two technical challenges. The first one is *how to incorporate and represent multimodal information* that also respects smart contract semantics during finetuning. Our evaluation shows that simple prompt engineering without customized training datasets cannot identify correct invariants in real-world contracts.

TABLE 4: SmartInv inferred invariants in Listing 3. 19+ and 25+ refer to the line-numbered location where the invariant should be inserted, e.g., 19+ means immediately after line 19. `Old(votingToken.balanceOf(address(this)))` returns the votingToken balance in `address(this)` before the instrumented function is called. At 19+, inferred invariant specifies that the current balance of `votingToken` equals to the sum of transferred amount and the prior balance before the transfer. At 25+, the inferred invariant specifies that total balance of `votingToken` stays the same after a proposal round has ended, i.e., no flashloan transfers into `votingToken`.

Critical Program Points	Inferred Invariants
19+	`assert(votingToken.balanceOf( address(this))== Old(votingToken.balanceOf( address(this)))+amount);`
25+	`assert(Old(votingToken.balanceOf( address(this)))==votingToken.balanceOf( address(this)));`

To this end, we use tailored invariant types (§3.2) and build a unique finetuning process (in §3.3) that incorporates multimodal information. Our finetuning process tailors answers to ToT prompts. Furthermore, foundation models are known to have the hallucination problem [42]. Thus, a second challenge is *to determine which invariants are correct* during inference on previously unseen contracts without ground truth. To overcome the second challenge, SmartInv adopts a new invariants ranking strategy for effective verification (in §3.4).

Finetuning and Ground Truth. Finetuning foundation models to both consistently reason about diverse sets of invariants and detect a wide range of real-world bugs is non-trivial. Given that no prior foundation-model-based work has done both (to the best of our knowledge), SmartInv presents the first multimodal-reasoning-based finetuning approach for invariant generation and bug detection.

SmartInv employs step-by-step reasoning following ToT with tailored ground truth to finetune the pre-trained model. Each training sample consists of a smart contract file collected from Etherscan [2] and annotated ground truth of six attributes (details in Table 5). The input contract, the ToT prompt, and the corresponding ground truth are encoded as token sequences for finetuning, following the next-token-prediction paradigm of the model. For example, a ToT prompt with ground truth answers can be "What are the critical program points in the contract? Critical program points are [ground truth]."

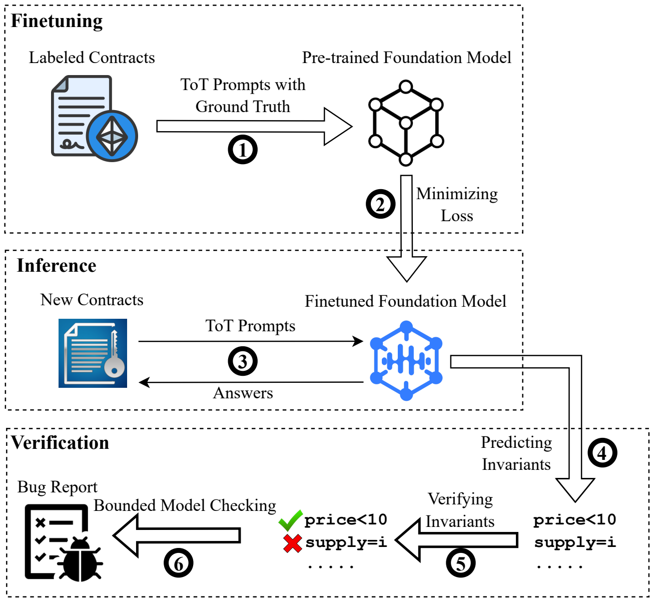

Figure 1: SmartInv's Workflow

Inference. The finetuning design above facilitates SmartInv's unique inference procedure – using iterative and increasingly complex prompts that respect smart contracts' domain-specific properties and semantics, e.g., transaction context and critical program points. During inference, the finetuned model is fed with a previously unseen contract and a tiered prompt without any answers, such as "What are the critical program points in the contract?" The finetuned model can generate critical program points as an answer because the model is finetuned for this specific downstream task. SmartInv then uses inferred answers from prior prompts to elicit answers on more challenging prompts.

Verifying Predicted Invariants. After inference, SmartInv ranks critical invariants for verification: it prompts the model to rank the inferred invariants from most likely to be correct and bug preventive to the least likely. After ranking, SmartInv automatically switches to a verifier that tries to prove program correctness based on ranked invariants. If such proofs can be found on an inferred invariant, SmartInv marks that invariant as correct. Otherwise, SmartInv uses a bounded model checker to search for violations of inferred invariants. When violations are found, such violations signify two scenarios that warrant further review: i) a potential bug; or ii) potentially incorrect invariants. Therefore, we inspect the counterexamples to i) confirm the existence of bugs and thereby correct invariants or ii) confirm the incorrectness of inferred invariants. If no correctness proofs or counterexamples are found, SmartInv discards that unproven invariant and moves on to the next invariant.

3. Methodology

In this section, we formally define the finetuning task for invariant inference. We then describe the invariant types we developed to facilitate the Tier of Thought (ToT) finetuning. Lastly, we describe how we verify the inferred invariants based on the ranked invariants from ToT prompting.

3.1. Problem Formulation

Let M_θ be the pre-trained foundation model parameterized by θ and let S be the tokenized input contract. Let C be tokenized program points (i.e., a line-numbered location of a code statement) and V be the space of invariants. we finetune M_θ to generate critical program points $c_i \in C$ and associated invariants $v_i \in V$ as a tuple (c_i, v_i). From the predicted (c_i, v_i), we finetune M_θ to predict the vulnerabilities.

The vulnerability prediction serves as an auxiliary task when the invariant verifier fails to find any counter example or correctness proof (§3.3).

3.2. Invariant Types

Broadly, SmartInv infers three types of invariants to capture functional bugs: assertions with special expressions, modifiers, and global invariants. These invariants are highly generalizable and can be integrated into state-of-the-art verifiers [1], [61], [80]. Table 5 illustrates the use of a subset of these invariant types on the simple smart contract shown in Listing 4.

A common invariant type inferred by SmartInv is `assert(expr1 op expr2)` at a critical program point. `expr1` and `expr2` are legal Solidity expressions. `op` are binary comparison operators, such as ==, >=, <=, !=. Assertions can also be replaced by Solidity's `require(expr, error_msg)`, as well as pre-condition check `Assume(expr)` and post-condition checks `Ensures(expr)` because SmartInv's backend verifier also supports these additional checks.

As part of assertions, SmartInv also infers special expressions uniquely tailored to smart contracts, such as `Old(expr)`, `k*Old(expr)`, and `SumMapping(mappingVar)`. The use of `Old(expr)` is similar to Daikon's `Orig()` [17]. It returns the value of `expr` before a function is called. Ratio `k` allows users to specify an accepted volatility ratio for a variable, usually a price point. SmartInv sets the default volatility ratio k to 2. Finetuning SmartInv to learn `Old(expr)` and `k` enables invariant inference related to inconsistent state updates and price manipulation bugs. To check arithmetic operations across mapping reference type (a multi-layered key value storage) and integer/byte primitive types, SmartInv sums up the values stored in multi-layered maps using customized `SumMapping(mappingVar)`, where `mappingVar` is a mapping type in Solidity that acts like a nested key-value hash table. Modifiers are invariant-like Solidity functions that specify the behavior of other functions. SmartInv infers function modifiers, because they are useful to express expected behavior of an entire function beyond assertions. Take Table 5 as an example. SmartInv infers an `onlyOwner` modifier at critical program point 10+. When the `tokenIncrease()` function at line 12

```
1  contract trainingExample {
2   /* totalSupply and balances should be same before and after
        transfer*/
3   uint totalSupply, tokens;
4   mapping(address ⟹ uint) balances;
5
6   function transfer(address to) external {
7    balances[to]= balances[to].add(tokens) ;
8    balances[msg.sender]= balances[msg.sender].sub(tokens) ;
9    }
10
11  /* only contract owner should invoke tokenIncrease*/
12  function tokenIncrease() external {
13   if (tokens <=100) {
14     tokens+=1.1*tokens;
15    }
16   return tokens;
17  }
18  ...
19 }
```

Listing 4: Training example contract

is instrumented with the modifier, only contract owner whose address is tracked by the `owner` state variable can invoke `tokenIncrease()` function. Intuitively, modifiers specify function-level behavior.

To specify cross-function and cross-contract behavior, SMARTINV also learn to infer `Invariant(expr)`, a function that specifies an invariant that will hold true for some duration of the smart contract execution as opposed to only at specific program points. It can be useful to check expected return values of a function during cross-contract calls. e.g., `Invariant(func()==a)`. Secondly, `Invariant(expr)` can specify loop invariants when placed at the beginning of loops. A third use case is to check the value range of a state variable x for entire smart contract execution by specifying e.g., `Invariant(x > 10)` outside functions in the contract.

3.3. Tier of Thought Finetuning and Inference

To guide a pre-trained foundation model towards generating bug-preventive invariants, the key innovation is to introduce increasingly complex thoughts to reason from contract source files to correct answers of each prompt. Given input contract, each thought is a tokenized sequence, such as "What is the transaction context in the contract? The transactional contract is token transfer."

Finetuning with ToT. We finetune the model to generate one thought at a time, starting with the simplest and working our way up to the most complex thoughts. Taking Listing 4 and Table 5 as an illustrative training example for this section, the model learns to reason about smart contracts' source code and natural language highlighted in blue (variable names and comments useful for program understanding and invariants generation). Using multimodal information, the model is finetuned to generate tokenized answers to a prompt. This design enables SMARTINV to predict domain-specific information (i.e., the ground truth of labeled features) on new contracts. SMARTINV adds "<end of text>" as a special token to separate each training sample.

TABLE 5: Labeled ground truth for the trainingExample contract. Repeated code fragments are replaced by ... in "Critical Invariants" and "Ranked Critical Invariants" labels. "Rank 1","Rank 2", and "Rank3" refer to a group of invariants that can discover bugs in descending likelihood.

Attributes	**Example Ground Truth**
transaction context	token transfer
critical program points	7+, 8+, 10+, 12, 17+
Invariants	7+ `assert(balances[msg.sender]>=tokens);` 8+ `assert(sumMapping(balances)== totalSupply);` 10+ `modifier onlyOwner{ require(msg.sender==owner);};` 12 `function tokenIncrease()` *`onlyOwner`* `external {...};` 17+ `Invariant(tokenIncrease()>100);`
Critical Invariants	7+ `assert(...);` 8+ `assert(...);` 10+ `modifier onlyOwner{...};` 12 `function tokenIncrease(uint tokens)` *`onlyOwner`* `external {...};`
Ranked Critical Invariants	Rank 1: 10+ `modifier onlyOwner{...};` 12 `function tokenIncrease()` *`onlyOwner`* `external {...};` Rank 1: 7+ `assert(...);` Rank 2: 8+ `assert(...);` Rank 3: 17+ `Invariant(...);`
Vulnerabilities	incorrect visibility/ownership; arithmetic flaw;

Tier 1 Finetuning (Critical Program Points). In this tier, the model tokenizes contract source files, tier 1 prompts, and answers from labeled ground truth as sequences. The tier 1 thoughts below seek to finetune SMARTINV's understanding of transaction contexts and critical program points from multimodal sources in the contract. The example below illustrates tier 1 training sample:

> `Contract trainingExample {...}`
> What's the transaction context of the contract? The transaction context is **token transfer**.
> Given transaction context, what are the critical program points? Critical program points are **7+, 8+, 10+, 12, 17+**.
> <End of Text>

Tier 2 Finetuning (Invariants). At tier 2, the model tokenizes contract source files, tier 2 prompts, and the ground truth of "Invariants" and "Critical Invariants" labels for finetuning. Critical invariants refer to those that are likely to prevent bugs. SMARTINV's tier 2 finetuning design continues the thoughts from tier 1 and facilitates invariants generation at correct program points during inference. The example in grey box illustrates the training sample design of tier 2:

```
Contract trainingExample {...}
```
Given inferred critical program points, what are the invariants? Invariants are [ground truth of “Invariants” label in Table 5].
Given inferred invariants, what are the critical invariants? Critical invariants are [ground truth of “Critical Invariants” label in Table 5].
<End of Text>

`assert(sumMapping(balances)== totalSupply)` at critical program point 8+ in Table 5 is derived from both natural language cues and source code. This invariant checks that the condition specified in natural language at line 2 holds true. SMARTINV uses a special expression `sumMapping(*)` to sum up all balances stored in the `balances` mapping. This way, SMARTINV can directly compare `balances` of mapping type with `totalSupply` of uint type. Another natural language inspired invariant is `modifier onlyOwner(...)` at critical program point 10+, with added `onlyOwner` modifier instrumentation to the `tokenIncrease` function at line 12. This pair checks that only the contract owner can invoke `tokenIncrease` function as hinted by comments at line 11. By the invariant pairs, SMARTINV learns natural language hints and associated invariants during finetuning.

The remaining ground truth of the “Invariants” label in Table 5 are based on source code. The invariant `assert(balances[msg.sender])>= tokens)` at program point 7+ checks that a message sender has enough balance to make the transfer. `Invariant(tokenIncrease()>100)` at program point 17+ checks that the return value of `tokenIncrease` function is greater than 100 during cross-contract calls.

After being finetuned to generate invariants, SMARTINV is also finetuned to generate critical invariants that can potentially identify bugs. The invariant `assert(balances[msg.sender])>= tokens)` checks against arithmetic flaws. The invariant `modifier onlyOnwer{...}` with function signature instrumentation checks against incorrect access control under incorrect visibility/ownership (IVO) bug category in Table 2. Thus they are labeled as critical invariants for Listing 4 training contract. By comparison, the invariant at 17+ `Invariant(tokenIncrease()>100)` checks function return value without likely bugs in this sample. Therefore, 17+ `Invariant(...)` is not included in the ground truth of “Critical Invariant” label. With critical invariants, SMARTINV learns to reason about likely invariants that can guard vulnerable code fragments effectively.

Tier 3 Finetuning (Ranked Invariants and Bugs). SMARTINV is also finetuned to rank (prioritize) critical invariants and predict vulnerabilities in the contract from previously generated information. The example below illustrates the construction of tier 3 training sample:

```
Contract trainingExample {...}
```
What are the ranks of inferred critical invaraints? The ranks of inferred critical invariants are [ground truth of ”Ranked Critical Invariants” label in Table 5].
What are the vulnerabilities in the contract? The vulnerabilities are [ground truth of “vulnerabilities” label in Table 5].
<End of Text>

Specifically for Listing 4, the first rank 1 invariants at critical program points 10+ and 12 identify an incorrect visibility/ownership bug. The second rank 1 invariant at critical program points 7+ identifies an arithmetic flaw bug: the contract lacks proper guard to ensure that a message sender has sufficient balances to make a token transfer. Invariants of rank 2 and rank 3 are correct but trivial invariants that are less likely to find bugs.

Inference with ToT. The unique aspect of SMARTINV’s inference is to decompose the invariant inference problem into three-tiered tasks, thereby making an iterative process. SMARTINV uses inferred answers from the previous tier to generate answers for more challenging prompts of later tiers. At each tier, SMARTINV prompts the finetuned model on a previously unseen contract with two tailored prompts.

At first, SMARTINV tackles the simple task by inferring transaction contexts and critical program points. SMARTINV uses the answer generated from prompt A to continue generating critical program points asked by prompt B.

Tier 1 Prompts
Prompt A: What’s the transaction context of the contract?
Prompt B: Given transaction context, what are the critical program points?

At tier 2, SMARTINV is given a slightly more complex task of inferring invariants at predicted critical program points and identifying critical invariants, i.e., bug-preventive invariants from all inferred invariants.

Tier 2 Prompts
Prompt A: Given inferred critical program points, what are the invariants?
Prompt B: Given inferred invariants, what are the critical invariants?

At tier 3, SMARTINV is given the most challenging task: ranking critical invariants for verification and predicting vulnerabilities in a contract. SMARTINV reports verified invariants with predicted vulnerabilities as a final report. We highlight that buggy traces from a verifier are more sound than model inferred bugs. However, because verifiers frequently encounter incompatible Solidity compilers, SMARTINV reports predicted vulnerabilities as a remedy.

Tier 3 Prompts
Prompt A: What are the ranks of inferred critical invaraints?
Prompt B: What are the vulnerabilities in the contract?

3.4. ToT Invariants Verification Algorithm

Algorithm 1 has three phases to verify inferred invariants from ToT: candidate invariants ranking; proving program

Algorithm 1 ToT Invariants Verification

```
Input: an input smart contract $S$, Tier 3 invariants ranking model $M_\theta$,
       and an initial set of assumed true positive candidate invariants
       $V$ generated by $M_\theta$.
Output: verification report $P$, a set of verified correct invariants $I_{correct}$,
        and a set of possibly correct invariants that require further
        inspection $I_{possible}$.
1:  $Syntax\_Check(V)$;
2:  while $V \neq \emptyset$ do
3:      $v_i \leftarrow$ Tier3.rank();                        ▷ ranking critical invariants
4:      $V \leftarrow V - v_i$;
5:      $\Pi(\sigma(v_i), \mu)$ = $Inductive\_Check(S, v_i)$;
6:      if $\sigma(v_i) == True$ then          ▷ proof of program correctness found
7:          $I_{correct}.append(v_i)$;
8:          $P = P.append(\Pi)$
9:      else                                ▷ proof of program correctness not found
10:         $\Pi(\phi, \overline{\mu})$ = $BMC(S, v_i, m)$;             ▷ entering BMC phase
11:         $P = P.append(\Pi)$;
12:         if $\overline{\mu}$ is a counter example then
13:             $I_{possible}.append(v_i)$;
14:             break;  ▷ manual inspection requested for counterexamples
15:         end if
16:         discard $v_i$;   ▷ no counterexamples or correctness proof found
17:     end if
18: end while
19: return $P$, $I_{correct}$, and $I_{possible}$;
```

correctness by induction; bounded model checking. S denotes an input contract; V denotes a set of candidate invariants; v_i denotes a tier-3 ranked critical invariant selected from V; $I_{correct}$ denotes verified correct invariants; $I_{possible}$ denotes possibly correct invariants that require human inspection. From a set of candidate invariants V and an input contract S, the algorithm discards the invariants that cause compilation errors at line 1 first. Then tier 3 ranks critical invariants from candidate invariants at line 3. The novelty of Algorithm 1 is using foundation model guidance (critical invariants ranking) for verification, and as a result, increases verification efficiency. When generating bug reports, SMARTINV prioritizes buggy traces from the verifier. However, as this algorithm is limited by compatitable Solidity compilers, SMARTINV uses model inferred vulnerabilities as a last resort.

Induction Phase. After SMARTINV unrolls ranked invariants at line 3 using tier 3's prompt A, the $Inductive\_Check()$ function at line 5 maps Solidity to Boogie and uses Boogie's monomial predicate abstraction [39], [43] to check whether v_i is inductively strong enough to prove program correctness. SMARTINV selects Boogie for two specific features: i) it has built-in map types, e.g., [int] bool. These map types correspond well to Solidity's mapping types, e.g., mapping(int=>bool). Smart contracts use mappings frequently. ii) Boogie produces failing traces for each buggy procedure that can be fed into the next phase for precisely tracking on what kind of input triggers invariant violations.

If an invariant is inductively strong enough, Boogie will generate a proof of correctness μ. In this case, SMARTINV adds v_i to $I_{correct}$ as verified correct invariants and the full verification result Π is added to the final report P. If an invariant is not inductively strong enough, the bounded model checker will search for counterexamples.

Bounded Model Checking Phase. If invariant v_i cannot be proven inductive, i.e., $\sigma(v_i)$ is false, our bounded model checker $BMC(...)$ leverages CORRAL [40] to search for counterexamples $\overline{\mu}$. If counterexamples are found on v_i, there are two possible cases: i) v_i is a correct (true positive) invariant and a bug is found by the merit of counterexamples; or ii) v_i is an incorrect invariant. Therefore, we break the loop for further inspection. This algorithm is applied iteratively until all ranked critical invariants are evaluated.

4. Implementation

We implemented SMARTINV's training and inference in 4,011 lines of Python and the invariant verification algorithm in 1,322 lines on top of VERISOL [80].

Model Optimization and Hardware. We selected LLaMA-7B [78] as the backbone of SMARTINV. To enable memory-efficient training, we applied 8-bit quantization [23], Parameter Efficient Finetuning (PEFT) [45], and low-rank adaptation (LoRA) [34] to LLaMA-7B during finetuning. This optimization allowed our model to complete training within a single Nvidia RTX 2080Ti GPU, as opposed to usual requirements of 4 A6000 GPUs.

We ran all experiments and evaluations on a Linux Server with Intel Xeon 4214 at 2.20GHz with 48 virtual cores, 188GB RAM, and 4 Nvidia RTX 2080Ti GPUs, a Google coLab plus account with additional computation units, and a commercial server with 4 A6000 GPUs.

Dataset. We collected source files of 179,319 contracts in total, covering a period from January 1, 2016 to July 1, 2023. Of those 179,319 contracts, 175,991 contracts were crawled from Etherscan [2] via Google BigQuery and 3,328 contracts were crawled from 78 live decentralized applications (dApps)' public Git repositories. We selected 572 contracts (2,173 annotated samples post ToT data augmentation) that represented each bug type in Table 2 for training.

For evaluation, we excluded 89,698 contracts that are: i) duplicates; ii) require old Solidity compilers ($<$ 0.3.x); iii) written in non-Solidity languages (Vyper and Go); iv) already included in our training dataset. Thus, our evaluation dataset consists of 89,621 Solidity contracts averaging 1,621 lines of code per contract and they are different from our training dataset. Following [10], we categorized our evaluation dataset into three subsets based on lines of code: i) *small*: [0, 500); ii) *medium*: [500, 1000); iii) *large*: [1000, ∞), consisting of 65,739, 12,011, and 11,871 contracts respectively.

Labeled Attributes. To provide domain-specific insights, we labeled the ground truths of each training contract with six attributes: transaction contexts, critical program points; all relevant invariants; critical invariants, ranked critical invariants; vulnerabilities if the contract contains any. As illustrated in §3, we embedded the ground truth via ToT prompts during finetuning. Specifically, we have labeled contracts with the following ten transaction contexts in our training dataset: ERC libraries, token transfer, cross bridge, bidding, voting, lottery, healthcare, investing, price oracle, and others. These labels cover the top use cases identified by Ethereum [30]. To ensure correct labeling of ground truths, we ran the verification algorithm in §3.4 and cross-checked with at least two researchers. To ensure correct vulnerability labels, we reproduced 3,213 hacks in 4,033 lines of Solidity on a forked Ethereum virtual machine (EVM) [73] to confirm the existence of labeled vulnerabilities.

Hyperparameters. For model optimization, we set LoRA_alpha = 32, lora_dropout = 0.01, LoRA_R = 8, learning rate = 3e-4, micro_batch = 1. During inference, we set temperature, top-k, top-p, and repeated penalty to 0. The hyperparameters of other three finetuned foundation models are documented in SMARTINV's Github README.

We also note that pre-trained LLaMA cannot process contracts beyond 20 lines due to limited token length. Finetuning breaks such limitation by adding additional token mappings from the initial 512 to 4096. Therefore, our finetuned LLaMA can reason about medium ([500, 1000) lines) contracts. On large ([1000, ∞) lines) contracts, finetuned LLaMA are still limited by available token length. In that case, we prompted the model to summarize imported modules, i.e., imported library and helper contracts, to fit in available tokens.

TABLE 6: Reported alarms breakdown by type from 89,621 contracts. The last seven rows report functional bugs. Alarms are not necessarily true positive or exploitable bugs.

Bug Type	SMARTINV	VERISOL	SMARTEST	VERISMART	MYTHRIL	SLITHER	MANTICORE
RE	9,011	1,591	0	0	1,311	2,533	901
IF	13,531	2,031	31,655	29,015	602	952	421
AF	11,009	905	10,921	12,548	648	0	421
SC	908	0	452	366	99	972	122
EL	611	0	0	0	82	1,200	34
IG	494	0	0	2	12	78	122
IVO	1,022	4,899	3,091	3,001	23	79	90
PM	2,651	5	2	0	0	0	0
PE	3,019	0	0	0	0	0	0
BLF	1,091	84	0	0	0	0	0
IS	977	33	5	5	107	0	0
AV	2,065	0	0	0	0	0	0
CB	3,192	0	0	0	0	0	0
IDV	1,924	0	0	0	0	0	0
Total Alarms	51,505	1,924	46,126	44,937	2,884	5,814	2,111

TABLE 7: Report on total count of internal error and timeout results from 89,621 contracts. SMARTINV reported model inferred results when the verifier was incompatible.

	SMARTINV	VERISOL	SMARTEST	VERISMART	MYTHRIL	SLITHER	MANTICORE
Error	0	18,769	7,859	11,859	14,211	63,807	23,301
Timeout	0	0	31,636	32,825	72,526	0	44,209

5. Evaluation

We evaluate SMARTINV to answer the following questions:

- **RQ1**: How does SMARTINV compare to prior smart contract bug analyzers?
- **RQ2**: How does SMARTINV compare to prior invariant inference tools?
- **RQ3**: How much do our selected model LLaMA and optimizing strategies improve the accuracy of bug detection and invariants generation?
- **RQ4**: How fast is SMARTINV compared to baseline tools?

Experiments Setup. To sufficiently represent available tools, we selected bug analyzers covering a wide range of techniques with minimal overlapping. We installed and followed the instructions of the latest versions (as of July 28, 2023) of each bug analyzer from their Git repositories, and reached out to the authors when we encountered errors.

Ground Truth Measurement. We define ground truths in two relevant aspects: bugs and invariants. For bug detection, we conducted both large-scale and refined experiments. The large-scale experiment summarized each tools' reported alarms. We further validated the reported alarms by manually reviewing a subset of projects in the refined experiment. For invariants generation, we inspected the invariants generated in the refined experiments to gain a granular understanding. The scale of our analysis and ground truth measurement are in line with previous work [10], [69], [70].

Each tool under evaluation scanned the 89,621 contracts in our large-scale experiment, and we recorded their reported bug alarms in Table 6. We acknowledge that results in Table 6 summarize each tool's reported alarms, which may not necessarily be true positive or exploitable bugs. To gain more insights into each tool's false positives/negatives, we conducted a refined experiment on 60 well-known hacked projects (1,241 buggy contracts) and used their audit reports as ground truths to validate bug alarms. We recorded correct (TP), incorrect (FP), and missed (FN) alarms on bugs in the contract. To determine the ground truth of the detected bugs by each model, we defined "Accuracy (Acc.)" on a per-contract basis: we marked an output as accurate only when a model generated bug-preventive invariants at correct program points for an entire contract and inferred the correct bugs.

We reported three outcomes on each contract and grouped the results by bug type: i) *alarms*: the number of alarms of a given bug type; ii) *error*: a tool aborted due implementation issues; for example, VERISMART and SMARTEST require annotated test oracles to reasons about contracts; VERISOL and INVCON are not compatible with Solidity compilers $\geqslant$ 0.7.x; iii) *timeout*: a tool failed to produce alarms within a 30-minute time budget.

We evaluated the invariant generation on a per-invariant basis. That is, we manually inspected each generated invariant and considered it as accurate if it captured the correct properties without syntactical errors. We note that an accurate invariant can be trivial, meaning that correct invariants do not always prevent bugs.

5.1. RQ1: Bug Detection

We studied SMARTINV's bug-detection scope in the context of six similar prior tools: i) VERISOL(as is); ii) VERISMART [70], a CEGIS-style verifier; iii) SMARTEST [69], a language-model guided symbolic execution tool; iv) MYTHRIL [15], a commercial symbolic execution tool; v) MANTICORE [52], a commercial symbolic execution tool; vi) SLITHER [18], a static analyzer. We also studied SMARTINV and three other prompting-based approaches [13], [16], [72].

TABLE 8: Refined bug detection analysis on sampled 1,241 real-world functional buggy contracts with report on correct (TP), incorrect (FP), and missed (FN) bug alarms. ✗: a tool did not produce any results. This table reported results pertaining to functional bugs only. An alarm is correct if a tool generates specifications and/or traces correctly pinpointing a bug.

Contracts	**SmartInv**			**VeriSol**			**SmarTest**			**VeriSmart**			**Mythril**			**Slither**			**Manticore**		
	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN	TP	FP	FN
hundredFinance	45	3	0		✗			✗			✗			✗		45	0	5		✗	
sherlockYields	31	5	0	1	1	3		✗			✗		3	1	0	30	0	4		✗	
dfxFinance	72	6	0	1	0	1		✗			✗			✗			✗			✗	
Bacon	92	0	0	1	0	1		✗			✗			✗			✗			✗	
AnySwap	91	0	0	1	0	0		✗			✗		2	1	5	9	1	6		✗	
Dodo	4	4	1	1	0	0	5	0	0	3	0	0	1	0	9	4	5	3		✗	
Dao	1	0	0		✗			✗			✗		1	0	2	2	2	1		✗	
Bancor	24	1	0		✗			✗			✗		1	0	9		✗			✗	
beanStalk	41	2	0		✗			✗			✗		1	0	10	4	2	3		✗	
BeautyChain	1	0	0		✗			✗			✗			✗		1	4	9		✗	
Melo	13	0	0		✗			✗			✗			✗		1	1	10	4	1	0
NoodleFinance	2	0	0	1	0	1	0	1	1	0	1	1	3	1	11	1	4	8	9	0	0
BGLD	2	0	0	1	0	4	4	0	0	4	0	0	0	1	23	1	4	2	3	0	0
GYMNetwork	1	0	0	2	0	3	2	0	0	2	0	0	1	1	20		✗		2	2	0
eslasticSwap	2	0	0	1	0	5	9	0	2	5	0	2	4	0	19	5	3	12	5	0	0
EulerFinance	14	13	0	5	0	13	1	0	5	1	0	5	5	0	41	7	2	1	6	1	0
Meter	15	0	0	3	0	4	9	0	9	9	0	9		✗		1	5	1		✗	
NXUSD	23	4	0	1	0	4	5	0	1	5	0	1		✗			✗			✗	
monoSwap	19	1	0	8	0	1	6	0	4	6	0	4		✗			✗			✗	
LIFI	18	0	0	0	✗		6	2	5	4	2	5		✗			✗			✗	
MuBank	14	0	0	9	0	0	0	1	0	0	1	0		✗		9	5	5		✗	
OneRing	12	0	0	5	0	0	0	5	0	0	5	0		✗		2	4	9		✗	
Paraluni	6	0	0	6	0	0	0	1	0	0	1	0		✗		1	1	8		✗	
InverseFinance	9	1	0	4	0	2	0	1	0	0	1	0		✗		1	2	4		✗	
nimBus	3	0	0	0	✗		0	1	0	0	1	0		✗		1	1	10	9	0	4
moneyReserve	4	2	0	3	1	2	0	1	0	0	1	0		✗		1	0	12	1	0	0
pancakeswap	18	7	0	7	1	7	0	2	1	0	2	1		✗			✗		3	1	1
uniswap	42	9	0	8	1	9	1	9	7	1	8	7		✗		3	1	3	2	1	1
visor	31	12	0	7	0	9	9	6	8	6	6	7		✗		2	8	15		✗	
DFX	2	9	0		✗		7	4	4	0	4	4	5	2	22		✗			✗	
Harvest	9	5	0	4	0	8	0	10	3	0	0	3	1	0	4	0	3	1		✗	
moon	13	3	0		0	7	2	0	9	2	0	9	1	0	19	1	2	4		✗	
VFT	2	2	0		✗		10	1	2	10	1	2	1	0	31		✗			✗	
proxyTransfer	4	1	0	0	1	1	1	0	8	1	0	8	1	0	6	1	0	13		✗	
Nomad	5	1	0	1	0	3	0	0	1	0	0	1	1	3	8	3	1	12		✗	
Fundstransfer	15	1	0	0	1	3	2	9	3	2	9	3	1	1	9	2	2	2	1	3	3
walnutFinance	23	1	1	0	1	3	3	0	1	3	0	1	0	0	1	5	2	5	1	2	5
Umbrella	19	1	0	1	0	3	5	0	0	5	0	0	1	1	5	9	2	4	1	1	10
Fortress Loan	5	2	0	1	0	3	1	8	0	1	8	0	1	0	7	0	2	3	1	2	3
ShadowFinance	4	0	0		✗		1	0	0	1	0	0	1	0	1		✗		1	1	3
FeiProtocol	9	9	0		✗		1	0	0	1	0	0	1	0	5		✗		1	7	1
Revest	10	3	0		✗		1	0	0	1	0	0	1	0	9		✗		1	7	1
Cartel	3	0	0		✗		1	0	0	1	0	0	1	0	13	2	7	3	1	8	3
Qubit	11	2	0		✗		1	0	0	1	0	0	1	0	11	1	2	8	1	2	5
ValueVaults	2	1	0		✗			✗			✗			✗		1	1	3		✗	
PancakeBunny	3	1	0		✗			✗			✗		2	0	5	1	0	3		✗	
Nomad	13	2	0		✗			✗			✗			✗		1	0	5		✗	
SandleFinance	25	1	0	0	1	5	5	6	2	5	6	2	0	4	9	1	0	1		✗	
bunnyswap	16	1	0	2	0	6	2	1	1	2	1	1	1	3	10	1	0	0		✗	
MonoX	13	1	0	3	0	3	10	2	0	10	2	0	6	2	19		✗	5		✗	
CreamFinance	52	0	0	0	4	6	9	9	1	9	9	1	1	1	3		✗	9		✗	
Jay	12	2	0	1	0	9	0	4	1	0	4	1	9	2	12		✗	7	1	3	1
sushiSwap	32	1	0		✗		0	5	3	0	5	3	1	1	4	3	4	2	0	5	3
polynetwork	40	2	0		✗			✗			✗		5	5	5	2	5	0	0	2	4
ChainSwap	18	3	0		✗		2	9	4	2	9	4	4	9	9	3	6	5		✗	
grimFinance	22	1	0		✗		9	1	1	9	1	1	1	0	14	1	5	2		✗	
Ragnarok	15	3	0	0	1	3	4	0	2	4	0	2	1	1	4	1	0	13		✗	
XSurge	41	0	0	5	0	1	6	1	1	6	1	1	3	5	13	4	0	0		✗	
templeDao	14	0	0	6	0	0		✗			✗		8	2	2	5	0	13		✗	
RariFinance	4	0	0	0	1	0		✗			✗		7	2	3		✗			✗	
BabySwap	5	0	0		✗			✗			✗		2	1	2	0	1	3		✗	
1241 contracts	1111	129	2	100	14	133	140	100	90	122	89	89	91	50	414	179	101	257	54	49	48
percentage	**90.25%**	**10.39%**	**0.3%**	8.12%	12.28%	20.59%	11.37%	41.67%	13.93%	9.91%	42.18%	13.78%	7.39%	35.46%	64.08%	14.54%	36.07%	39.78%	4.39%	47.57%	7.43%

Since [13], [16] do not have a named tool, we refer to [13] by their model in use as ChatGPT and refer to [16] by its abbreviated paper title as Manual_Audit. We refer to [72] by its tool GPTScan.

In our large-scale experiment, we ran each tool on the entire dataset of 89,621 contracts until a tool terminated or timed out after 30 minutes. We reported alarms by each bug type. In our refined experiment, we sampled 1,241 contracts from 60 hacked live projects in the last two years. To provide refined false positive and false negative analysis, we confirmed a total of 456 natural functional bugs from audit reports and injected 775 functional bugs for precise bug tracking, totaling 1,231 ground truth bugs in this experiment.

Bug Alarms. Table 6 shows that SmartInv, VeriSol(as is), SmarTest, VeriSmart, Mythril, Slither, Manticore reported alarms on 57.47%, 10.65%, 51.47%, 50.14%, 3.22%, 6.49% of the evaluated contracts. Compared to existing tools, SmartInv generated more alarms on 5,397 contracts with major performance gains from functional bugs, reporting 14,797 more functional bugs than existing tools. We acknowledge that bug alarms are proxies for potential bugs, which do not imply true positive or exploitable bugs. We reported SmartInv's alarms based on the verifier's traces and model-inferred vulnerabilities.

Table 7 summarizes the number of errors and timeouts of each tool. To optimize symbolic execution and model-

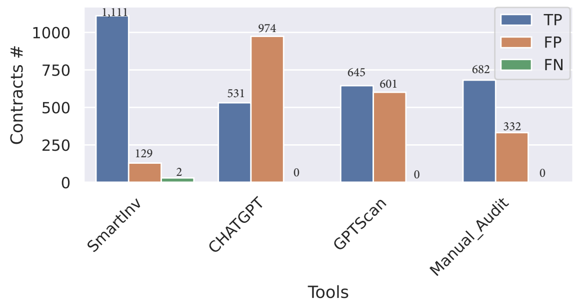

Figure 2: Comparison of SMARTINV with three prompting-based tools using the 1,241 contracts in the refined experiments of Table 8.

checking tools' performance, we increased their default memory budget to 10GB and disabled bounding if the option was available. The results of SMARTEST, VERISMART, MYTHRIL, and MANTICORE demonstrate limited scalability as a major drawback of symbolic execution tools, which had timeouts on at least 35% of evaluation contracts given the 30-minutes time budget. Verifiers' and static analyses' [18], [70], [80] errors were due to incompatible Solidity versions. SMARTINV did not report internal errors or timeouts, because the model could reason about contracts without requiring compiled source code.

False Positive Analysis. Table 8 summarizes refined bug detection analysis. SMARTINV, VERISOL, SMARTEST, VERISMART, MYTHRIL, SLITHER, MANTICORE had 10.39%, 12.28%, 41.67%, 42.18%, 35.46%, 36.07%, 47.57% false positive rate on functional bugs. SMARTINV was able to analyze 85% more contracts than prior tools (up to 1,057 more contracts), because the latter had timeouts and errors. This result was from manual review on bug alarms related to functional bugs only.

False positives of existing tools were largely due to matching of spurious patterns. For example, some [15], [52], [69], [70] mistook Listing 3 as an integer overflow (IF) bug. Some of SMARTINV's false positives resulted from bugs outside SMARTINV's detection scope. For example, on Fei Protocol contracts, SMARTINV's false positives were due to not recognizing incorrect function selector hashing, which SMARTINV mispredicted as arithmetic flaws (AF).

Listing 5 shows a common false positive result by SMARTINV. The first two functions `A (uint x)` and `A (bytes32 x)` have function selectors [71]: `0x2fbebd38` and `0xb42e8758`, which should be the case for different functions. The bug lies in the third function `A (uint x, uint y)`, which shares the same selector as `A (uint x)`. When external accounts call them, clashing selectors can cause either function to be called randomly. SMARTINV's false positives arose from not recognizing such clashing.

False Negative Analysis. SMARTINV missed only two bugs, with a false negative rate of 0.3%. Although the rationale behind foundation models' reasoning remains blackbox, one plausible explanation is that finetuning using the functional bugs improved the model's understanding of code semantics

```
function A(uint x) public returns (uint) {
        return x + 2;
    }
function A(bytes32 x) public returns (bytes32) {
        return keccak256(abi.encodePacked(x));
    }
function A(uint x, uint y) public returns (uint) {
        return x*y;
    }
```

Listing 5: SMARTINV's false positive patterns

TABLE 9: Invariants inference analysis. LOC (Avg.): average lines of code analyzed by a tool. #invariants/Contract: average invariants generated per contract. #FP/Contract: incorrect invariants generated per contract on average. An inferred invariant is considered incorrect if it cannot be provably verified or if it is inferred at wrong program points.

	SMARTINV	INVCON	VERISMART
LOC (Avg.)	**1,621**	862	354
# Invariants / Contract	6.00	**11.70**	3.00
#FP / Contract	**0.32**	2.41	0.92

under various contexts. Prior tools missed a large portion of functional bugs, because their heuristics were not designed for the purpose of functional bug detection.

Comparisons with Prompting Based Tools. We highlight that existing prompting frameworks do not involve finetuning, and that design choice accounts for major performance differences. Since CHATGPT and MANUAL_AUDIT do not offer open source implementation at the time of writing, we prompted the model by strictly following the format in the papers. As for GPTSCAN, we used the prompting templates in its Git repositories. Figure 2 demonstrates SMARTINV' effectiveness in localizing true positive bugs while minimizing false positives. CHATGPT, GPTSCAN, and MANUAL_AUDIT assume contracts under test are always buggy, thus no false negative results.

Overall, SMARTINV found 2×, 1.5×, and 1.5× more bugs than CHATGPT, GPTSCAN, and MANUAL_AUDIT, respectively. This gain was driven by SMARTINV' ability to detect functional bugs, as SMARTINV detected up to 216 more functional bugs than the others. SMARTINV reported a low false positive rate of 10.39%, lower than similar prompting frameworks. This result shows the drawbacks of prompting alone: without targeted finetuning and verification, model hallucination may lead to higher false results.

5.2. RQ2: Invariants Inference Accuracy

Invariants Inference Results. We studied SMARTINV's invariants generation in the context of INVCON, a Daikon-adapted smart contract invariants detector, and VERISMART, a CEGIS-style verifier. We obtained INVCON's results from its published webfront UI, and the scope was thus limited because INVCON's docker was no longer functioning. INVCON did not generate invariants with associated program points, so we manually reviewed and speculated the corresponding

TABLE 10: Finetuned candidate model evaluation (*GPT4 results were obtained from prompt engineering alone without finetuning on curated training dataset due to close source).

Model	Acc.	Prec.	Rec.	F1
Alpaca [74]	0.72	0.72	0.58	0.65
T5-Small [63]	0.81	0.81	0.70	0.75
GPT2 [25]	0.57	0.57	0.20	0.30
GPT4* [57]	0.44	0.43	0.32	0.19
OPT-350M [90]	0.53	0.53	0.25	0.34
LLaMA-7B [78]	**0.89**	**0.89**	**0.83**	**0.82**

program points. Table 9 reports SMARTINV's enhanced output after the check (Algorithm 1) and shows that on average, SMARTINV analyzed 88.05% and 357.90% more lines of code than INVCON and VERISMART per contract, because existing tools reported many internal errors.

False Positive Invariants Analysis. Each incorrectly inferred invariant was counted as a false positive. SMARTINV's overall false positive pattern was inferring correct invariants with respect to contracts at large, but at wrong program points. INVCON's false positives were due to imprecise mappings between Solidity and Java. VERISMART's false positive invariants came from incorrectly inferred transactional invariants. Although INVCON inherited Daikon's soundness, INVCON did not generate precise program points or reason about loop invariants. VERISMART was highly effective in generating invariants against integer overflow/underflow (IF) bugs, but VERISMART could not reason about invariants from multimodal information.

5.3. RQ3: Ablation Study

We considered six foundation models as our baselines: OPT-350M [90], Google's T5-Small [63], OpenAI's GPT2 [25] and GPT4 [57], Stanford's Alpaca [74], and Meta's LLaMA-7B [78]. Notably, GPT4 was not available for finetuning on customized datasets (as of July 31, 2023), so we supplied test contract source code and applied ToT prompting to GPT4 without finetuning. We finetuned the remaining five candidate models with multimodal information (without architectural optimization).

We quantified the effect of our key optimization strategies on end results. To evaluate each of SMARTINV's strategies, we removed natural language modality in the dataset by deleting implementation-related comments and renaming function and variable names without giving away domain-specific information, e.g., we renamed "votingToken" variable to "var." We removed ToT and ToT-related finetuning by using a one-shot general prompt as "What are the vulnerabilities and invariants in the contract?" For optimization, we compared SMARTINV' performance on unmodified LLaMA-7B against the architecturally optimized model.

Model Selection Results. Table 10 demonstrates that finetuned LLaMA outperformed T5-Small by 8% in accuracy and precision, 13% in recall, and 7% in F1. This observation implies scaling law (bigger models usually work better) [81] applies to smart contract invariant inference.

TABLE 11: Ablation study. Natural Language: natural language modality. ToT: tier of thought finetuning and prompting. Optimization: model architectural optimization. Full SMARTINV: no strategy removed and SMARTINV is at the default setting of source code and natural language modalities. Full SMARTINV with Tx. Hist.: "Tx. Hist." refers to deployed contracts' transaction history. SMARTINV with Tx. Hist. is finetuned on source code, natural language, transaction history modalities.

Remove	Acc.	Prec.	Rec.	F1
All	0.12	0.15	0.10	0.14
Natural Language	0.62	0.60	0.30	0.45
ToT	0.24	0.18	0.20	0.16
Optimization	0.89	0.88	**0.85**	0.82
Full SMARTINV	**0.89**	0.89	0.83	0.82
Full SMARTINV with Tx. Hist.	0.89	**0.92**	0.84	**0.85**

Effect of Natural Language Modality. Table 11 shows that once we removed natural language modality, the accuracy and F1 of SMARTINV dropped by 27% and 37%, respectively. SMARTINV's recall dropped from 83% to 30% once we removed natural language. We also examined the effect of natural language modality on a per-contract basis. We observed that both SMARTINV with natural language information and without it detected implementation bugs equally well. However, SMARTINV with natural language modality detected far (40×) more functional bugs than single-modal SMARTINV, with major performance gain from natural language modality.

Effect of ToT. Table 11 demonstrates that ToT had a significant impact: SMARTINV's accuracy, precision, recall, and F1 dropped by 65%, 71%, 63%, and 66% respectively when ToT was removed. Without ToT, SMARTINV repeated the invariants in the test contracts without generating bug-preventive invariants. This result shows that ToT was crucial to SMARTINV's bug detection performance.

Effect of Quantization and PEFT. Table 11 shows architecturally optimized SMARTINV achieved comparable results as SMARTINV without architectural optimization. SMARTINV with optimization and that without had the same accuracy of 89%. While SMARTINV with optimization outperformed that without by 1% in precision, the latter outperformed the former by 2% in recall. Both had the same F1 score of 94%, indicating that Quantization and PEFT did not drastically increase false positives or false negatives in the case of invariant inference. This result shows that architecturally optimized SMARTINV decreased the cost of finetuning GPU consumption by up to 75% without incurring significant accuracy loss.

Transaction History as Additional Modality. We also conducted additional experiments on using transaction history as an additional modality for deployed contracts. We incorporated transaction history that covered a wide range of

TABLE 12: Mean runtime analysis (in seconds) of each tool on evaluation dataset.

	Small	Medium	Large	Full
#Contracts	65,739	12,011	11,871	89,621
SMARTINV	15.02	32.98	37.77	28.59
VeriSol	232.51	1612.32	3933.21	2994.01
SmarTest	175.01	297.02	3908.32	2793.45
VeriSmart	27.21	33.76	4145.22	3105.40
Mythril	404.98	305.22	5031.33	3580.51
Slither	22.35	155.41	9080.62	3451.13
Manticore	301.33	562.21	7281.94	4715.16

bugs, as shown in Table 2 during finetuning and inference. Specifically, for finetuning, we crawled the transaction history of 329 deployed contracts from Etherscan and added them to the corresponding contracts in our training data. The included transaction history was users' addresses, call functions, and call data. For inference, we tested SMARTINV with transaction history on previously unseen deployed contracts in our evaluation dataset.

We modified prompt B in Tier 1 as "Given [transaction history] and transaction context, what are the critical program points?" With transaction history, SMARTINV's precision and F1 score in Table 11 improves by 2.9% and 3%, respectively. This improvement applies universally to a broad range of functional bugs. Transaction history can be helpful for SMARTINV to focus on critical and bug-prone functions. Given that SMARTINV's key strength is to detect bugs in smart contracts' source code pre-deployment, SMARTINV does not assume available transaction history modality by default. However, we expect that reasoning about transaction history can further improve SMARTINV's bug detection performance on deployed contracts.

5.4. RQ4: Runtime Performance

We ran each tool on the entire evaluation dataset and recorded the runtime accordingly. Since VERISOL requires manual specification, we thus approximated the manual effort of VERISOL by interviewing three seasoned auditors at two separate smart contract audit companies [33], [68]. We learned that each real-world contract takes an experienced auditor from 1 hour to 3 hours. This leads to an average of 90 minutes per contract to construct the manual specification for VERISOL.

Table 12 shows the average runtime on a per-contract basis. SMARTINV benefits from the use of GPUs and can complete inference within about 30 seconds. Notably, SMARTINV's runtime overhead does not increase by more than 17 seconds when the size of contracts increases by 500 lines of code. As an average speedup compared to manual audits or human reviews, SMARTINV saves auditors by up to 78 minutes on large contracts.

6. Discussion and Limitations

Token Length. Foundation models are known to have limited input token length. For example, LLaMA is limited to 4,096 tokens (approximately 3000 words after tokenization). Therefore, large contracts with more than 2,000 lines are often cut short in the reasoning process. We take initial steps towards addressing this by prompting the model to summarize imported modules in a contract under test. This approach introduces unsound elements into SMARTINV' invariant inference process. For future work, we plan to incorporate other promising strategies such as retrieval augmented prompting [14].

Verifier Compatibility with Solidity Compilers. We design our verifier based on VERISOL's mappings between Solidity and Boogie. VERISOL is limited to Solidity compiler between 0.4.0 and 0.7.0. Our verifier also inherits the limitation of VERISOL's. A large number of contracts do not have compatible compiler versions. In that case, we manually reviewed SMARTINV inferred invariants and vulnerabilities. We plan to expand our verifier across newer Solidity compiler versions in future work.

Exploitabiltiy of Zero-Day Bugs. Not all detected zero-day bugs are exploitable. For instance, integer overflow/underflow (IF) bugs only exist in contracts built on older Solidity compilers ($\leqslant$ 0.6.0). New Solidity compilers automatically check for over/underflow and thus preempt such exploitability. Newly upgraded proxy contracts also prevent exploitable zero-day bugs.

Threats to Validity. Our results were obtained on selected evaluation dataset, which might not be representative of newer contracts. Secondly, we did not report results based on bugs' exploitability and did not compare SMARTINV with other tools in that regard. Evaluation based on exploitable bugs may be different. Thirdly, SMARTINV does not discard implied trivial invariants and this can be further optimized in future work. Lastly, despite of our best effort to be precise and accurate, manual inspection on SMARTINV's inferred vulnerabilities and manual classification of reported specifications into true and false positives are inherently challenging and can be subjective in some cases.

Ethical Vulnerability Disclosure. When we discovered bugs, we *never* executed or attempted to test them on deployed smart contracts. Instead, we tested them in a forked local environment. When disclosing zero-day bugs, we ensured to first consult relevant developers if possible and anonymized contract code and addresses. Notably, entities and individuals behind some deployed smart contracts are unknown. In that case, we reported the bugs to Common Vulnerabilities and Exposure (CVE) Database.

7. Case Study: Zero-Day Bugs

SMARTINV detected 119 zero-day bugs. We manually confirmed each bug by launching a successful exploit in a sandboxed clone of the real blockchain environment [6], [73]. Five of the bugs occur in smart contracts listed on Immunefi [35], a bug bounty website, so we reported the bugs to this website. All of them were confirmed as "high severity" and fixed quickly, earning us a bounty of $17,600. Due to the pseudonymity of the blockchain, only a small number of

```
1  contract Bridge {
2    function init(
3      uint32 _callSite,
4      address _sender,
5      bytes32 _merkleRoot
6       ) public {
7          base_initialize(_sender);
8          callSite = _callSite;
9          committedRoot = _merkleRoot;
10         //invariant #1: assert(_merkleRoot != 0);
11         confirmAt[_merkleRoot] = 1;
12      }
13 ...
14
15  function process(bytes memory _message)
16     public returns (bool _success) {
17     ...
18     //zero day vulnerability
19     //invariant #2: assert(_msgHash != 0);
20     assert(accept(messages[_msgHash]));
21     }
22
23   function accept(bytes32 _root)
24     public view returns (bool) {
25     //invariant #3: assert(_root != 0);
26     uint256 _time = confirmAt[_root];
27      }
```

Listing 6: Anonymized contract code snippet

these contracts have associated developer information, so we sampled three more bugs and reported them. One of them was quickly fixed. We received no replies from the other two bugs' developers, most likely because the projects were no longer maintained. This section provides two example bugs. We anonymized the contract in §7.1 at request.

7.1. Cross Bridge

Listing 6 provides a buggy code snippet of a cross bridge contract. SMARTINV discovers the zero-day vulnerability related to the assumed unique `_msgHash` at line 20. Since the default value of unknown `_msgHash` is `0x00` in Solidity, this bug can potentially greenlight a malicious actor's `_msgHash` value that defaults to `0x00` during cross-bridge communication. As a result, the malicious actor can bypass the assertion check at line 20, where `messages[0x00]` is an acceptable root. SMARTINV detects this vulnerability with invariant `assert(_msgHash != 0)` to prevent incorrect default values.

7.2. Inefficient Gas

Listing 7 contains a gas inefficient `remove()` function that can lock users' funds, leading to denial of services to users. A user can join the vault by first joining `DepositQueue`. While in the queue, users can choose to refund their deposit or to process it at the end of a transaction round. However, line 7 uses a gas-expensive for-loop implementation to remove each deposit on the queue. A long queue can easily exceed the 30 million per block gas limit even with a single deposit operation [65]. An attacker can send funds from different accounts to occupy the queue. As a result, the contract will lose the ability to refund or

```
1  abstract contract BaseVault {
2
3      DepositQueueLib DepositQueue;
4
5      function processQueuedDeposits(uint256 startIndex,
           uint256 endIndex) external {
6          uint256 _totalAssets = totalAssets();
7          for (uint256 i = startIndex; i < endIndex; i
               ++){
8              uint256 currentAssets = _totalAssets +
                   processedDeposits;
9              depositEntry = depositQueue.get(i);
10             processedDeposits += depositEntry.amount;
11         }
12         //invariant #1: require(depositQueue.size()
               ==1, "Cannot process multiple deposits");
13         depositQueue.remove(startIndex, endIndex);
14     }
```

Listing 7: Code snippets of baseVault.sol

process users' deposits because all deposits are locked in `DepositQueue` due to insufficient gas.

SMARTINV discovers this bug by generating an invariant after line 12 to check the size of `DepositQueue`. The inferred invariant `require(depositQueue.size() == 1, "Cannot process multiple deposits")` ensures that there is only one deposit on the queue each time. This specified property is important for safeguarding the `remove()` function from running out of gas.

8. Related Work

Smart Contract Static and Dynamic Analysis. SMARTEST [69], MYTHRIL [15], MANTICORE [52], MAIAN [55], TEETHER [38], and ETHBMC [20] are symbolic execution tools that generate vulnerable transaction sequences. SMARTEST utilizes language models to supplement symbolic execution on smart contracts. MAIAN and TEETHER focus on high-level bugs such as Ether-leaking and suicidal vulnerabilities. ETHBMC focuses on memory modeling and cryptographic hashing. They largely rely on pattern specific heuristics to detect certain classes of implementation bugs.

SLITHER [18], OYENTE [50], OSIRIS [76], and HONEYBADGER [77] are static analysis tools that utilize data flow analysis. They analyze the source code of a smart contract to identify potential security vulnerabilities. Static analysis cannot reason about mutlimodal input. As a result, they are limited in their abilities to detect functional bugs.

Fuzzing is also common dynamic analysis used by smart contract security researchers. CONFUZZIUS [75], ECHIDNA [26], FLUFFY [85] and SFUZZ [53] are recently developed smart contract fuzzers. They send random inputs to a smart contract and try to trigger unexpected behavior and identify potential security vulnerabilities. Fuzzing-based tools tend to be slow, because they need to explore many possible transactional states of a contract.

Invariants Detectors and Verifiers. One popular program analysis approach is invariants detection. DAIKON [17] and INVCON [48] are both invariants detection tools. Daikon does not apply to Solidity and INVCON is a DAIKON-adapted tool that maps Solidity to Java. SMTCHECKER [7], SOLC-VERIFY [32], VERISOL [80], and ZEUS [36] are verification frameworks that require manually specifying invariants first and then automatically infer transaction invariants during verification. Manual specification is often error-prone.

ML-Based Invariant Inference and Bug Detection. ML-based tools include SVCHECKER [87] and NEURAL CONTRACT GRAPH [93]. They use deep neural networks to discover limited sets of implementation bugs such as reentrancy based on general patterns. ESCORT [49] is based on bytecode level transfer learning and ETH2VEC [8] uses language models to detect code clones. Existing prompting-based tools [13], [16], [59], [72] have achieved impressive results even witthout finetuning. Prior ML-based tools mainly focus on implementation bugs, because they often rely bug-specific graph search heuristics. Recently, using foundation models to generate invariants [60] has been proposed as a promising direction in other programming languages such as Java. This direction, along with SMARTINV, hopefully presents a new way forward in the context of ML-based approaches.

9. Conclusion

This paper introduced SMARTINV, an automated framework for detecting both implementation and functional bugs in smart contracts. SMARTINV can reason across multiple modalities of smart contracts, including source code and natural language, and reason over them based on a new prompting strategy called Tier of Thoughts (ToT) to generate bug-preventive invariants. Evaluation of SMARTINV on real-world contracts revealed SMARTINV performed well on both invariants generation and bug detection tasks.

Acknowledgement

We would like to express our sincere appreciation to Jianan Yao for his invaluable guidance and advice on this project, as well as to Andreas Kellas, Chengzhi Mao, and Zhuo Zhang for their extensive edits and feedback. We also extend our gratitude to the anonymous reviewers and our shepherd for their constructive comments, which significantly improve this paper. This work was supported in part by Columbia Center for Digital Finance and Technologies and gifts from Google, Accenture, and DiDi.

References

[1] Crytic safety properties: https://github.com/crytic/properties.

[2] Etherscan: https://etherscan.io/.

[3] Gemma strategies: https://github.com/gammastrategies/uniswapv3-risk-mitigation/blob/main/notes%20on%20uniswap%20v3 %20risk%20mitigation.md.

[4] Smart contract security field guide: https://scsfg.io/hackers/oracle-manipulation/.

[5] Visor attack address: 0x10c509aa9ab291c76c45414e7cdbd375e1d5ace8. .

[6] Foundry toolchain. 2023.

[7] Leonardo Alt and Christian Reitwiessner. SMT-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation*. Industrial Practice, 2018.

[8] Nami Ashizawa, Naoto Yanai, Jason Paul Cruz, and Shingo Okamura. Eth2vec: learning contract-wide code representations for vulnerability detection on ethereum smart contracts. In *Proceedings of the 3rd ACM International Symposium on Blockchain and Secure Critical Infrastructure*. BSCI'21, 2021.

[9] Davide Balzarotti. The use of likely invariants as feedback for fuzzers. In *30th USENIX Security Symposium. USENIX Security'21*, 2021.

[10] Priyanka Bose, Dipanjan Das, Yanju Chen, Yu Feng, Christopher Kruegel, and Giovanni Vigna. Sailfish: Vetting smart contract state-inconsistency bugs in seconds. In *2022 IEEE Symposium on Security and Privacy*. IEEE, 2022.

[11] Lexi Brent, Neville Grech, Sifis Lagouvardos, Bernhard Scholz, and Yannis Smaragdakis. Ethainter: a smart contract security analyzer for composite vulnerabilities. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[12] David Brumley, Tzi-cker Chiueh, Robert Johnson, Huijia Lin, and Dawn Song. Rich: Automatically protecting against integer-based vulnerabilities. 2007.

[13] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. When chatgpt meets smart contract vulnerability detection: How far are we? *arXiv preprint arXiv:2309.05520*, 2023.

[14] Xiang Chen, Lei Li, Ningyu Zhang, Xiaozhuan Liang, Shumin Deng, Chuanqi Tan, Fei Huang, Luo Si, and Huajun Chen. Decoupling knowledge from memorization: Retrieval-augmented prompt learning. *Advances in Neural Information Processing Systems*, 35, 2022.

[15] Consensys/mythril, 2022. https://github.com/ConsenSys/mythril.

[16] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338*, 2023.

[17] Michael D Ernst, Jeff H Perkins, Philip J Guo, Stephen McCamant, Carlos Pacheco, Matthew S Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Science of computer programming*, 69(1-3), 2007.

[18] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE, 2019.

[19] Visor Finance. Post-mortem for vvisr staking contract exploit and upcoming migration. *Medium:https://medium.com/visorfinance/post-mortem-for-vvisr-staking-contract-exploit-and-upcoming-migration-7920e1dee55a*, 2021.

[20] Joel Frank, Cornelius Aschermann, and Thorsten Holz. Ethbmc: A bounded model checker for smart contracts. In *Srdjan Capkun and Franziska Roesner*. 29th USENIX Security Symposium, USENIX Security, 2020.

[21] Liam Frost. Defi token visr plunges by 95 *Crypto Briefing: https://cryptoslate.com/defi-token-visr-plunges-by-95-following-8-million-hack/*, 2021.

[22] Asem Ghaleb and Karthik Pattabiraman. How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[23] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[24] Github. Timelockcontroller vulnerability in openzeppelin contracts. March 2022.

[25] OpenAI GPT2. Better language models and their implications. March 2022.

[26] Gustavo Grieco, Will Song, Artur Cygan, Josselin Feist, and Alex Groce. Echidna: effective, usable, and fast fuzzing for smart contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[27] Alex Groce, Josselin Feist, Gustavo Grieco, and Michael Colburn. What are the actual flaws in important smart contracts (and how can we find them)? In *Financial Cryptography and Data Security: 24th International Conference, FC 2020, Kota Kinabalu, Malaysia, February 10–14, 2020 Revised Selected Papers 24*. Springer, 2020.

[28] Bishwas C Gupta, Nitesh Kumar, Anand Handa, and Sandeep K Shukla. An insecurity study of ethereum smart contracts. In *Security, Privacy, and Applied Cryptography Engineering: 10th International Conference, SPACE 2020, Kolkata, India, December 17–21, 2020, Proceedings 10*. Springer, 2020.

[29] Mudit Gupta. Visor finance hack proof of concept. *Github link: https://gist.github.com/maxsam4/91704944a5d7b5923649ba7752f18f1a*, 2021.

[30] Srajan Gupta. 10 real world use cases for ethereum. 2021.

[31] Seungwoong Ha and Hawoong Jeong. Discovering invariants via machine learning. *Physical Review Research*, 3(4).

[32] Ákos Hajdu and Dejan Jovanović. solc-verify: A modular verifier for solidity smart contracts. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*, page 1. Springer, 2020.

[33] Chainlink: https://blog.chain.link/how-to-audit smartcontract. March.

[34] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021.

[35] Immunefi, 2022. https://immunefi.com/.

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

[37] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *arXiv preprint arXiv:2205.11916*, 2022.

[38] Johannes Krupp and Christian Rossow. Teether: Gnawing at ethereum to automatically exploit smart contracts. In *27th $USENIX$ Security Symposium*, 2018.

[39] Shuvendu K. Lahiri and Shaz Qadeer. Complexity and algorithms for monomial and clausal predicate abstraction. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[40] Akash Lal and Shaz Qadeer. Powering the static driver verifier using corral. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, New York, NY, USA, 2014. Association for Computing Machinery.

[41] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *2001 USENIX Security Symposium, Washington, DC*, 2001.

[42] Minhyeok Lee. A mathematical investigation of hallucination and creativity in gpt models. *Mathematics*, 11(10), 2023.

[43] K Rustan M Leino. This is boogie 2. *manuscript KRML*, 178(131), 2008.

[44] Chao Liu, Han Liu, Zhao Cao, Zhong Chen, Bangdao Chen, and Bill Roscoe. Reguard: finding reentrancy bugs in smart contracts. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, 2018.

[45] Haokun Liu, Derek Tam, Mohammed Muqeeth, Jay Mohta, Tenghao Huang, Mohit Bansal, and Colin A Raffel. Few-shot parameter-efficient fine-tuning is better and cheaper than in-context learning. *Advances in Neural Information Processing Systems*, 35, 2022.

[46] Junrui Liu, Yanju Chen, Bryan Tan, Isil Dillig, and Yu Feng. Learning contract invariants using reinforcement learning. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[47] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9), 2023.

[48] Ye Liu and Yi Li. Invcon: A dynamic invariant detector for ethereum smart contracts. In *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[49] Ning Lu, Bin Wang, Yongxin Zhang, Wenbo Shi, and Christian Esposit. Neucheck: A more practical ethereum smart contract security analysis tool. In *Software: Practice and Experience*, vol. 51, no. 10, 2021.

[50] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16, 2016.

[51] Alexander Mense and Markus Flatscher. Security vulnerabilities in ethereum smart contracts. In *Proceedings of the 20th international conference on information integration and web-based applications & services*, 2018.

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

[53] Tai D Nguyen, Long H Pham, Jun Sun, Yun Lin, and Quang Tran Minh. sfuzz: An efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020.

[54] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th annual computer security applications conference*, 2018.

[55] Ivica Nikolic, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*. ACSAC '18, 2018.

[56] Jeremy W Nimmer and Michael D Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electronic Notes in Theoretical Computer Science*, 55(2), 2001.

[57] OpenAI. Gpt4: https://openai.com/gpt-4. 2023.

[58] OpenZeppelin. Timelockcontroller. *CVE-2021-39167, Common Vulnerabilities and Exposures Database*, 2021.

[59] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning*, pages 27496–27520. PMLR, 2023.

[60] Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. Can large language models reason about program invariants? In *International Conference on Machine Learning*. PMLR, 2023.

[61] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. In *2020 IEEE symposium on security and privacy (SP)*. IEEE, 2020.

[62] Kudelski Security Research. The polynetwork hack explained. August 2022.

[63] Adam Roberts and Colin Raffel. Exploring transfer learning with t5: the text-to-text transfer transformer. March 2022.

[64] Gabriel Ryan, Justin Wong, Jianan Yao, Ronghui Gu, and Suman Jana. Cln2inv: learning loop invariants with continuous logic networks. *arXiv preprint arXiv:1909.11542*, 2019.

[65] Openzeppelin Security. Pods finance ethereum volatility vault audit #1. March 2022.

[66] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*. Springer, 2017.

[67] Maheswar Sharma, Keerthana Kasthuri, Parvinder Singh, and Nynisha Akula. Smart contract vulnerabilities, attacks and auditing considerations. In *The Auditor's Guide to Blockchain Technology*. CRC Press.

[68] Cypher Shield. How long does smart contract audit takes. March 2023.

[69] Sunbeom So, Seongjoon Hong, and Hakjoo Oh. Smartest: Effectively hunting vulnerable transaction sequences in smart contracts through language model-guided symbolic execution. In *USENIX Security Symposium*, 2021.

[70] Sunbeom So, Myungho Lee, Jisu Park, Heejo Lee, and Hakjoo Oh. Verismart: A highly precise safety verifier for ethereum smart contracts. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[71] Solidity by example: Function selector.

[72] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. When gpt meets program analysis: Towards intelligent detection of smart contract logic vulnerabilities in gptscan. *arXiv preprint arXiv:2308.03314*, 2023.

[73] SunWeb3Sec. Defihacklabs git repository: https://github.com/sunweb3sec/defihacklabs/projects?query=is

[74] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. March 2022.

[75] Christof Ferreira Torres, Antonio Ken Iannillo, Arthur Gervais, and Radu State. Confuzzius: A data dependency-aware hybrid fuzzer for smart contracts. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2021.

[76] Christof Ferreira Torres, Julian Schütte, and Radu State. Osiris: Hunting for integer bugs in ethereum smart contracts. In *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018.

[77] Christof Ferreira Torres, Mathis Steichen, and Radu State. The art of the scam: Demystifying honeypots in ethereum smart contracts. In *Proceedings of the 28th USENIX Conference on Security Symposium*. 2019.

[78] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. arxiv.org, 2023.

[79] Shuai Wang, Chengyu Zhang, and Zhendong Su. Detecting nondeterministic payment bugs in ethereum smart contracts. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA), 2019.

[80] Yuepeng Wang, Shuvendu K Lahiri, Shuo Chen, Rong Pan, Isil Dillig, Cody Born, Immad Naseer, and Kostas Ferles. Formal verification of workflow policies for smart contracts in azure blockchain. In *Verified Software. Theories, Tools, and Experiments: 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13–14, 2019, Revised Selected Papers 11*. Springer, 2020.

[81] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022.

[82] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022.

[83] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[84] Siwei Wu, Dabao Wang, Jianting He, Yajin Zhou, Lei Wu, Xingliang Yuan, Qinming He, and Kui Ren. Defiranger: Detecting price manipulation attacks on defi applications. *arXiv preprint arXiv:2104.15068*, 2021.

[85] Youngseok Yang, Taesoo Kim, and Byung-Gon Chun. Finding consensus bugs in ethereum via multi-transaction differential fuzzing. In *OSDI*, pages 349–365, 2021.

[86] Jianan Yao, Gabriel Ryan, Justin Wong, Suman Jana, and Ronghui Gu. Learning nonlinear loop invariants with gated continuous logic networks. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[87] Ye Yuan and TongYi Xie. Svchecker: a deep learning-based system for smart contract vulnerability detection. In *Proceedings Volume 12260, International Conference on Computer Application and Information Security*. ICCAIS, 2021.

[88] Jiashuo Zhang, Jianbo Gao, Yue Li, Ziming Chen, Zhi Guan, and Zhong Chen. Xscope: Hunting for cross-chain bridge attacks. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ICSE'22, 2022.

[89] Pengfei Zhang, Huitao Shen, and Hui Zhai. Machine learning topological invariants with neural networks. *Physical review letters*, 120(6).

[90] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. arxiv.org, 2022.

[91] Zhilu Zhang and Mert Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *Advances in neural information processing systems*, 31, 2018.

[92] Zhuo Zhang, Brian Zhang, Wen Xu, and Zhiqiang Li. Demystifying exploitable bugs in smart contracts. In *Proceedings of nternational Conference on Software Engineering*. ICSE'23, 2023.

[93] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. IJCAI'20, 2020.

Appendix A. Meta-Review

The following meta-review was prepared by the program committee for the 2024 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

The paper presents SMARTINV, a smart contract invariant inference framework to automate the detection. The key insight of the paper is that the expected behavior of smart contracts, as specified by invariants, relies on understanding multimodal information, such as source code and natural language. Thus, SMARTINV combines the analysis of source code and natural language document to detect bugs in smart contracts. It uses a tiered prompting strategy to identify invariants by applying machine learning on source code and relevant comments and documents. These invariants are then used to detect bugs, particularly functional bugs. SMARTINV is evaluated relatively thoroughly on bug detection, invariants generation, and performance.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field.

A.3. Reasons for Acceptance

- The paper develops Tier of Thought (ToT), a general prompting strategy, that can be used to fine tune and elicit explicit reasoning of foundation models on the program structures of smart contracts.
- SMARTINV extracts invariants for expected behaviors of smart contracts by leveraging foundation models to reason about multimodal information including source code and natural language documents.
- The comprehensive experiments demonstrate SMARTINV's superiority over the state-of-the-art approaches in invariance inference.