SmartDecompiler-R1: Enhancing Faithful and Explainable Smart Contract Bytecode Decompilation with Reinforcement Learning

YILUN MA, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
LINGXIAO TANG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
LI LIN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
ZHIPENG GAO, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIACHI CHEN, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
XIN XIA, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
LINGFENG BAO*†, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

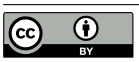
Understanding EVM bytecode is critical for smart contract security analysis. Existing decompilers typically rely on heuristic rules or leverage large language models (LLMs) to generate source code after bytecode analysis. However, heuristic-based approaches often produce pseudocode that is difficult for humans to interpret, while LLM-based methods also face several problems. LLMs have little experience with low-level bytecode, which weakens their reasoning and causes inaccurate results. Additionally, their tendency to auto-correct code breaks faithfulness to the original program. A general lack of clarity in these approaches also hinders effective auditing and interpretation. In this paper, we propose SmartDecompiler-R1, an end-to-end decompilation framework that translates Three-Address Code (TAC), a register-based representation of EVM bytecode, into source code using reinforcement learning. SmartDecompiler-R1 significantly improves both the accuracy and consistency of decompiled code, while additionally providing human-readable explanations for the bytecode-to-source generation process. To the best of our knowledge, we are the first to design a benchmark equipped with well-defined test cases and a systematic evaluation framework for smart contract decompilation. Experimental results on this benchmark demonstrate that SmartDecompiler-R1 substantially outperforms existing decompilers on execution consistency by 46.23%. In particular, compared with LLM-based approaches, SmartDecompiler-R1 achieves a significant improvement in preserving vulnerability consistency between the original and decompiled contracts. Furthermore, we conduct extensive ablation studies to validate the effectiveness of individual components of SmartDecompiler-R1, and design targeted experiments to demonstrate the interpretability and readability of the generated explanations.

CCS Concepts: • **Security and privacy → Software reverse engineering**.

Additional Key Words and Phrases: Smart Contract, EVM bytecode, Decompilation, Reinforcement Learning

*Also with Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security
†Corresponding author

Authors' Contact Information: Yilun Ma, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, 22421277@zju.edu.cn; Lingxiao Tang, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, 12421037@zju.edu.cn; Li Lin, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, linli1210@zju.edu.cn; Zhipeng Gao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, zhipeng.gao@zju.edu.cn; Jiachi Chen, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, chenjiachi@zju.edu.cn; Xin Xia, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, xin.xia@acm.org; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, Zhejiang, China, lingfengbao@zju.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA166
https://doi.org/10.1145/3832257

ACM Reference Format:
Yilun Ma, Lingxiao Tang, Li Lin, Zhipeng Gao, Jiachi Chen, Xin Xia, and Lingfeng Bao. 2026. SmartDecompiler-R1: Enhancing Faithful and Explainable Smart Contract Bytecode Decompilation with Reinforcement Learning. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA166 (October 2026), 22 pages. https://doi.org/10.1145/3832257

1 Introduction

Ethereum has grown rapidly as a widely used blockchain platform due to its support for self-executing smart contracts. Developers leverage these contracts to construct decentralized applications (i.e., DApps), facilitating trustless and disintermediated interactions. Smart contracts are developed in multiple programming languages, with the majority deployed on the Ethereum Virtual Machine (EVM) or EVM-compatible platforms. Solidity [14] and Vyper [50] are widely used for developing these contracts. When deploying smart contracts, developers need to compile the source code into bytecode before uploading it to the Ethereum blockchain. As noted in a previous study [56], 87% of deployed smart contracts on Ethereum lack publicly accessible source code, which creates significant challenges for auditing and analysis. A recent study [35] also demonstrates that widely-used source code verification platforms, such as Etherscan [15] and Blockscout [6], do not ensure consistency between verified source code and its corresponding deployed bytecode.

Due to the prevalence of scenarios where source code is unavailable, the analysis of bytecode becomes essential. Recently, decompilation [11, 19, 20, 29, 44, 64] for smart contract bytecode has been extensively studied to assist developers and auditors in better understanding bytecode. Previous research on decompilers has primarily focused on rule-based analysis of bytecode, employing heuristic algorithms to generate Solidity-like code [19, 20]. Nevertheless, the generated code is still far from human-readable. More recent work has started to integrate large language models (LLMs) into EVM decompilation pipelines in two ways: (i) post-editing and refining the outputs produced by heuristic decompilers [29], or (ii) synthesizing source-level code from an intermediate, logic-preserving description of the bytecode [44]. In both cases, LLMs are not fed raw bytecode; instead, they depend on a front-end analysis—typically heuristic decompilation and static analysis—to extract a structured representation that the LLM can reliably consume. For example, SmartHalo [29] augments heuristic decompilation with static-analysis signals to improve readability, whereas DiSCo [44] converts extracted bytecode semantics into natural-language descriptions for code generation. However, such LLM-assisted decompilation pipelines face three major problems:

Problem#1. The difficulty of general LLMs' reasoning ability over smart contract bytecode. Because general-purpose LLMs are trained on data that lacks sufficient bytecode-relevant information, most of them exhibit a limited understanding of bytecode semantics. Although existing decompilation techniques can produce preliminary pseudocode or natural language descriptions that capture the underlying bytecode logic, directly feeding such representations into LLMs remains suboptimal. This is because LLMs are primarily designed for source code and natural language, and thus lack sufficient exposure to inputs that explicitly encode fine-grained data-flow and control-flow semantics close to bytecode execution. As a consequence, a semantic gap emerges between bytecode-oriented representations and the abstraction level at which LLMs operate, limiting their ability to effectively reason about and reconstruct original smart contracts.

Problem#2. LLMs tend to sanitize or "auto-fix" vulnerabilities, which conflicts with the goal of faithful decompilation. Su et al. [44] find that some inconsistencies arise because LLMs tend automatically fix vulnerabilities when generating code. We also observe that, in the code decompiled by SmartHalo, LLMs introduce checks that are absent from the original function. While the safety-oriented generation behavior of LLMs is advantageous for smart contract generation, it is basically misaligned with the goals of decompilation. Decompilation aims to reconstruct a faithful and functionally equivalent representation of the original program, including the original

program's defects and vulnerabilities. When vulnerabilities are implicitly fixed or omitted during LLM-assisted decompilation, the decompiled code fails to faithfully reflect the original program, undermining decompilation as a tool for bytecode-level vulnerability analysis [10, 30, 31].

Problem#3. Existing decompilers provide limited transparency, making the bytecode-to-source mapping hard to interpret. Prior decompilation approaches mainly focus on generating high-level source code, while the underlying bytecode analysis is not exposed to end users. As a result, auditors must rely on the decompiled code alone to reason about bytecode-level semantics, which remains challenging for security analysis. Although some studies provide summaries for bytecode behavior [56], these summaries are often too coarse-grained. They do not establish clear mappings between bytecode instructions and reconstructed source-level code, thereby limiting interpretability and practical usability.

In this paper, we propose SmartDecompiler-R1, an end-to-end smart contract decompilation framework that translates EVM bytecode (via Three-Address Code, TAC [5, 7]) into human-readable source code while preserving security-relevant semantics. To achieve this, SmartDecompiler-R1 integrates dataset construction with a two-stage training pipeline: we first perform supervised fine-tuning on paired TAC–source functions to ground the model in bytecode-level understanding, and then apply GRPO-based reinforcement learning [21]. The reinforcement learning stage leverages execution-derived and vulnerability-related rewards to improve semantic faithfulness and vulnerability preservation, and we incorporate chain-of-thought [54] supervision to enhance transparency and interpretability of the bytecode-to-source reconstruction process. Next, we present the **Solution to Problem #1–#3** to explain how SmartDecompiler-R1 addresses each problem.

Solution to Problem #1. To reduce the semantic gap between bytecode-oriented representations and LLM, we first apply data-flow and control-flow analysis to transform stack-based smart contract bytecode into a register-based representation. This transformation makes implicit stack semantics explicit, exposing fine-grained data and control dependencies. Nevertheless, due to the lack of domain-specific knowledge of bytecode semantics, structural transformation alone is insufficient for effective LLM understanding. We therefore construct a supervised dataset that maps three-address code (TAC) to corresponding source-level functions, enabling the model to learn the semantics of TAC-level data-flow and control-flow. Through supervised training, the model aligns bytecode-level representations with high-level program abstractions, thereby acquiring the ability to understand bytecode semantics and generate accurate decompiled smart contract code.

Solution to Problem #2. To address this problem, we introduce a reinforcement learning–based optimization objective that explicitly encourages vulnerability preservation during decompilation. Specifically, we augment the reward function with an additional component that rewards the model for consistently reconstructing vulnerability-related program structures in the original source code. This reward design removes the implicit incentive for the model to repair or sanitize unsafe semantics during decompiling. Consequently, the model learns to faithfully reproduce security-relevant control flows and data dependencies. This approach realigns the decompilation objective with the requirements of vulnerability analysis, enabling SmartDecompiler-R1 to generate code that accurately reflects both the functional behavior and the defects of the original smart contract.

Solution to Problem #3. To address this problem, we synthesize chain-of-thought (CoT) explanations using LLM and use them as supervision during fine-tuning to improve interpretability. During reinforcement learning, the model is encouraged to generate CoT explanations alongside code predictions, which can be reused at inference time to explain the generated code. Given the assumed correlation between CoT explanation quality and decompilation accuracy, the model can improve its explanatory capability through reinforcement learning without explicitly rewarding explanations. Empirically, this training strategy also leads to more consistent and informative

reasoning traces, suggesting a strong correlation between decompilation accuracy and quality of the associated explanations.

To accurately evaluate the capabilities of SmartDecompiler-R1, we construct a dedicated benchmark for function-level smart contract decompilation. For each function in the benchmark, we automatically generate appropriate test cases using LLMs. These test cases provide a reliable and rigorous mechanism to verify whether the decompiled function exhibits execution behavior identical to the original function under the same inputs. Moreover, we design a set of reliable, multi-dimensional evaluation metrics, including BLEU-4, compilability, execution consistency, and vulnerability consistency, to assess whether the decompiled functions preserve semantic and behavioral consistency with their original counterparts. These metrics enable a fine-grained evaluation of decompilation quality at the function level from multiple perspectives. In addition, we provide a supporting testing framework to facilitate the application of these metrics and to ensure reproducible and consistent evaluation.

Experimental results demonstrate that SmartDecompiler-R1 significantly outperforms existing baselines in decompiling EVM bytecode at the function level. Compared with the strongest existing learning-based baselines, SmartDecompiler-R1 achieves improvements of **46.23%** on dynamic execution consistency. In addition, SmartDecompiler-R1 surpasses the best available open-source LLM-assisted baselines by **177.25%** on dynamic execution consistency. To further understand the sources of these improvements, we conduct ablation experiments to validate the contribution of each training stage and to highlight the critical role of reinforcement learning in enhancing the semantic consistency of decompiled functions.

In summary, we make the following contributions:

- We propose a novel framework, SmartDecompiler-R1, for function-level smart contract bytecode decompilation, which leverages GRPO-based reinforcement learning to guide the generation of semantically consistent high-level code from EVM bytecode.
- We construct a comprehensive benchmark accompanied by automatically generated test cases and a unified evaluation framework, providing a reproducible and extensible foundation for future research on smart contract decompilation.
- We enable the decompilation process to produce high-quality, fine-grained explanatory descriptions, offering clear and interpretable insights into how low-level bytecode semantics are translated into high-level decompiled functions.
- Through extensive experimental evaluation, we demonstrate the accuracy of the decompiled code generated by SmartDecompiler-R1, and show that the generated explanations substantially facilitate the understanding of EVM bytecode.

2 Background and Motivation

In this section, we first introduce the prerequisite background necessary to understand our decompilation pipeline, and then present a motivating example to illustrate the design motivation behind SmartDecompiler-R1.

2.1 Background

2.1.1 Smart Contract and EVM bytecode. Smart contracts are self-executing agreements implemented on blockchain platforms, a concept originally proposed by Nick Szabo [46]. Given their significant value, extensive research has been conducted to improve smart contract development, including approaches presented in [17, 26, 59]. On Ethereum, smart contract logic is executed in the form of EVM bytecode, which defines a low-level, stack-based instruction set executed uniformly

Bytecode with Explanation by SmartDecompiler-R1

```
//Block 0x531-0x53d:Function Entry & Validation
//This validates that no ether is being sent with the function call
0x531[ V0 = CALLVALUE; V1 = ISZERO V0; //V1 = ISZERO V0 checks if CALLVALUE is zero
       V2 = 0x53d; JUMPI 0x53d V1] -> 0x539,0x53d
0x53d[ V3 = 0x40; V4 = M[0x40];...; V7 = CALLDATALOAD 0x4; V8 = ADD 0x4 V7;
       V9 = CALLDATALOAD V8;...; V21 = 0xbfe; JUMP 0xbfe]
//Block 0xbfe-0xc16:Owner Permission Check
//This ensures only the contract owner can execute this function.
0xbfe[ V22 = 0x0; V23 = S[0x0]; V24 = 0x1;...; V30 = CALLER;
       V31 = EQ V30 V29;//V31 = EQ V30 V29 Compares caller address with owner address.
       V32 = 0xc16; JUMPI 0xc16 V31] -> 0xc12,0xc16
//Block 0xc1a-0xc70:Loop Initialization
//Loop iterates through addresses in _addresses array until all elements are processed.
0xc1a[ V34 = M[V4]; //V34 = M[V4] Loads array length from calldata
       V35 = LT S0 V34; //V35 = LT S0 V34 Compares loop counter with array length
       V36 = ISZERO V35; V37 = 0xc70; JUMPI 0xc70 V36] -> 0xc24,0xc70
0xc24[ V39 = M[V4]; V40 = LT S0 V39;..; V41 = ISZERO V40; JUMPI 0xc36 V42] -> 0xc35,0xc36
//Block 0xc36-0xc1a:Array Element Processing
//Processes each address element by storing it in memory and updating the mapping.
0xc36[ V44 = 0x20; V45 = MUL 0x20 S0; V46 = ADD V45 V4; V47 = ADD 0x20 V46;V48 = M[V47];
       ...; M[0x0] = V54;//M[0x0] = V54 Stores address at memory location 0x0
       ...; V58 = 0x0; V59 = SHA3 0x0 0x40; V60 = S[V59];...;
       V66 = OR V65 V63; S[V59] = V66;//S[V59] = V66 Updates mapping value in storage
       V68 = ADD 0x1 S5; V69 = 0xc1a; JUMP 0xc1a]//JUMP 0xc1a Continues loop iteration.
  ......
```

Ground Truth

```
event FreezeIn(address[] indexed from, bool value);
function setFreezeIn(address[] addrs, bool value)
 public onlyOwner {
  for (uint i=0; i<addrs.length; i++){
    freezeIn[addrs[i]]=value;
  }
  FreezeIn(addrs, value);
}
```

Decompiled by SmartDecompiler-R1

```
event SetFreezeIn(address[], bool);
function setFreezeIn(address[] _addresses,bool _freeze)
public onlyOwner {
  for (uint i=0;i<_addresses.length;i++) {
    freezeIn[_addresses[i]] = _freeze;
  }
  SetFreezeIn(_addresses, _freeze);
}
```

Fig. 1. Generated Results of SmartDecompiler-R1 and Ground Truth.

across the network. High-level smart contract languages such as Solidity are compiled into EVM bytecode before deployment, ensuring deterministic execution on all Ethereum nodes.

2.1.2 Three-Address Code. Three-address code (TAC) serves as a crucial intermediate representation that bridges high-level source code and low-level bytecode. EVM bytecode follows a stack-based execution model, in which instruction operands are implicitly determined by stack operations. This implicit operand representation obscures data dependencies in the raw bytecode and complicates direct data-flow analysis. By contrast, TAC makes data dependencies explicit by representing each instruction with clearly defined operands and results.

Most existing decompilation tools rely on this form of emulated execution to recover instruction operands. Several decompilation tools, such as Elipmoc [20] and GigaHorse [19], explicitly generate a TAC representation. Others, like Panoramix [1], perform an implicit bytecode-to-TAC translation as an intermediate stage without an explicit TAC construction phase. Capturing data flow from raw bytecode through execution emulation is relatively straightforward for symbolic or rule-based analyses, yet it is non-trivial for neural networks to infer such execution-dependent relationships directly. Given that a substantial body of prior work [2, 3, 5, 7, 19] has already focused on recovering TAC or TAC-equivalent representations from bytecode, we consider the transformation from bytecode to TAC as a largely solved problem. Accordingly, our work concentrates on the subsequent and more challenging step: transforming TAC into readable and high-level source code.

2.1.3 LLM Synthetic Data. Learning-based approaches rely heavily on high-quality datasets for effective supervised training. As a result, LLMs for dataset construction have attracted increasing attention in the software engineering domain. For program repair, Pham et al. [37] synthesize a repository-level bug repair dataset to enhance the repair ability of LLMs. For Smart Contract, Chen et al. [9] employ LLM to construct a Smart Contract vulnerability dataset. In binary analysis domains, CLAP [51] and BinQuery [61] use LLM to generate ground truth for supervised training. They proved the efficiency of synthesizing datasets with LLMs.

2.2 Motivating Example

Decompiling EVM bytecode into human-readable source code is essential for smart contract auditing, debugging, and program understanding. However, existing approaches still struggle to

Decompiled by Dedaub

```
function setFreezeIn(address[] addrs, bool value) public nonPayable {
    v0 = new uint256[](addrs.length);
    MEM[64] = 32 + (v0 + (addrs.length << 5));
    v1 = msg.data.length;
    CALLDATACOPY(v0.data, addrs.data, addrs.length << 5);
    require(msg.sender == _owner);
    v2 = v3 = 0;
    while (1) {
        if (v2 >= v0.length) {
            v4 = v5 = 0;
            while (v4 < v0.length << 5) {
                MEM[v4 + v6.data] = MEM[v4 + v0.data];
                v4 += 32;
            }
            MEM[v6.data] = value;
            emit FreezeIn(keccak256(v6.data), value);
            exit;
        } else {
            assert(v2 < v0.length);
            _freezeIn[address(v0[v2])] = value;
            v2 += 1;
        }
    }
}
```

Decompiled by Dedaub with LLM

```
event FreezeIn(address[], bool);
function setFreezeIn(address[] memory addrs, bool value) public {
 require(msg.sender == owner);
 for (uint256 i = 0; i < addrs.length; i++) {
  freezeIn[addrs[i]] = value;
 }
 emit FreezeIn(keccak256(abi.encodePacked(addrs, value)), value);
}
```

Decompiled by SmartHalo

```
event FreezeIn(address[] indexed from, bool value);
function setFreezeIn(address[] calldata addrs, bool value) external {
 require(addrs.length > 0, "Address array cannot be empty");
 require(msg.sender == contractOwner, "Only owner can call this function");
 for (uint256 i = 0; i < addrs.length; i++) {
  freezeInStatus[addrs[i]] = value;
 }
  emit FreezeIn(addrs, value);
}
```

Fig. 2. Decompilation Results by Dedaub and two LLM-based Approaches.

produce readable and trustworthy outputs. The limitations of current methods directly manifest the three core problems outlined in our introduction.

A common solution for EVM bytecode decompilation is heuristic rule-based translation from bytecode patterns to high-level code. While efficient and deterministic, it often yields low-level artifacts (e.g., excessive temporaries, unnatural control flow), leading to code that is hard for humans to interpret. Figure 1 shows the original bytecode and source code of the function `FreezeIn`, along with the output generated by SmartDecompiler-R1. This function is relatively simple, consisting mainly of a loop and an event emission. However, as shown in Figure 2, the code decompiled by Dedaub [12], a heuristic rule-based compiler, strictly follows the low-level control logic of the bytecode. Although this output correctly preserves the original control and data flow, it results in a complex and difficult-to-read decompiled function. This exemplifies the root cause of **Problem#1**: the raw bytecode representation is too far from the high-level abstractions that humans (and general-purpose LLMs) can understand, making direct or naive translation ineffective.

To mitigate this, recent LLM-based approaches like DiSCo [44] and SmartHalo [29] attempt to post-process or regenerate code from heuristic outputs. As shown in Figure 2, LLM-based post-processing can significantly simplify the structure and improve readability. However, this introduces **Problem#2**. Since these LLMs are not specifically trained for the decompilation task, they may misinterpret certain patterns or, more critically, "auto-fix" the code. For instance, SmartHalo introduces unnecessary checks during the decompilation process. These superfluous checks alter the original program's logic and may hinder the identification of actual vulnerabilities, directly conflicting with the goal of faithful decompilation.

Furthermore, all these baseline methods suffer from **Problem#3**: limited transparency. An auditor looking at the decompiled code from Dedaub, DiSCO, or SmartHalo is left to wonder why the code looks the way it does. There is no clear, step-by-step rationale connecting specific bytecode instructions to the high-level constructs in the output. This lack of a verifiable mapping makes it hard to trust the decompilation result for critical security analysis.

In contrast, Figure 1 shows that SmartDecompiler-R1 can almost completely recover the original source function. The only minor discrepancy lies in the naming of temporary variables, which does not affect function semantics. Crucially, SmartDecompiler-R1 also provides a detailed explanation of its reasoning process. This explanation highlights the key bytecode instructions that determine high-level semantics while filtering out low-level noise. It clearly maps blocks of bytecode to high-level concepts like "non-payable check," "ownership validation," and "loop structure." This

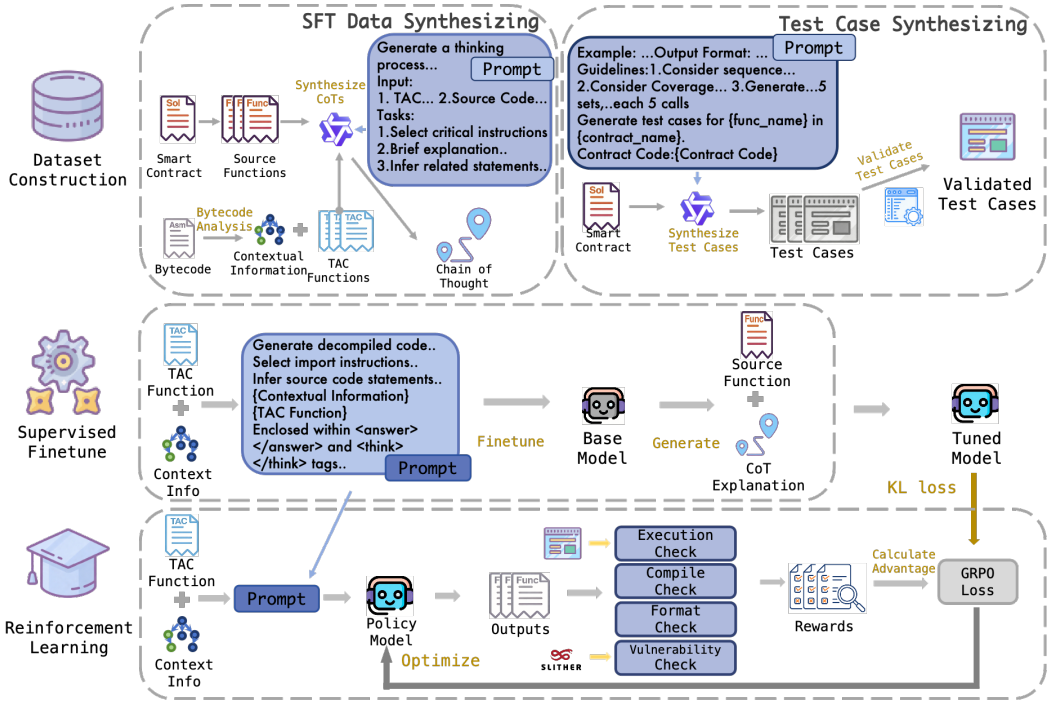


Fig. 3. Main Framework

directly addresses **Problem#3** by offering full transparency and interpretability, allowing auditors to verify the decompilation's correctness at a granular level.

3 Approach

Figure 3 illustrates the core components of SmartDecompiler-R1, which is structured into three main stages. In the first stage, we construct two high-quality datasets. One is a supervised fine-tuning (SFT) training dataset, created by synthesizing chain-of-thought reasoning traces with the LLM. The other is a dynamically executable dataset, built by generating test cases for smart contracts; this dataset is further partitioned into a reinforcement learning (RL) training set and a final test set. Second, we employ SFT to equip the model with the ability to explain the assembly code and decompile smart contract bytecode. Finally, we apply reinforcement learning to further refine the model's decompilation ability, shifting the optimization objective from purely syntactic accuracy toward the accurate preservation of security vulnerabilities and consistent runtime semantics.

3.1 Dataset Construction

3.1.1 SFT Data Synthesizing. SFT data synthesizing consists of two parts: bytecode preprocessing and chain of thought explanation synthesis. Bytecode preprocessing aims to extract bytecode-level functions and convert them into TAC, which explicitly represents data flow. It also infers contract-level information such as state variables and intra-contract function calls. Chain-of-thought explanation synthesis aims to leverage TAC functions and source-level functions to generate decompilation reasoning traces, which serve as chain-of-thought labels for supervised fine-tuning.

Bytecode preprocessing: To enable function-level decompilation, we must first extract individual functions from raw bytecode and enrich them with necessary global context including state variables

and inter-procedural relationships. We perform a three-stage preprocessing pipeline that transforms raw EVM bytecode into structured, context-augmented Three-Address Code (TAC) functions.

- **Function Extraction and TAC Generation.** We first disassemble the EVM bytecode using Vandal [7] and perform combined data-flow and control-flow analysis. This analysis recovers the function boundaries within the bytecode and explicitly represents data dependencies through TAC. The output of this stage is a set of TAC representations for each function-like segment in the bytecode.
- **Distinguishing Genuine Functions from State Variables**. Not all function-like bytecode segments correspond to actual source-level functions; some represent compiler-generated accessors for state variables. To distinguish between these two cases, we train a binary classifier based on a self-attention architecture, leveraging a Byte-Pair Encoding (BPE) [42] tokenizer pre-trained by CodeT5[52]. The classifier takes tokenized TAC functions and abi as input and predicts whether a segment corresponds to a genuine function or a state variable accessor. This model achieves 98% precision in identifying state variables.
- **Global Context Enrichment**. For segments classified as state variables, we recover their storage layout information. Specifically, we extract the accessed storage address from each storage operation. If the address is a constant, we treat it as a direct state variable access. Otherwise, we perform backward slicing on the address variable until it resolves to either a SHA-derived value (indicating indexing into a dynamic array) or an affine form $c + k$ (indicating indexing into a static array). This allows us to reconstruct the storage slot mapping for state variables.
  For genuine functions, we conduct inter-procedural block-level analysis to capture cross-function relationships. We observe that basic blocks shared across multiple functions often indicate either direct calls to other public functions or shared calls to private functions. While such block reuse can also result from compiler optimizations (e.g., code deduplication), we rely on the neural model's ability to learn characteristic function entry patterns (such as local variable initialization) to implicitly disambiguate true inter-procedural calls during training, which has been shown to be feasible in [23, 45].

The final output of our bytecode preprocessing is a collection of TAC functions, each augmented with relevant global context (including state variable mappings and cross-function call information), ready for supervised fine-tuning.

Chain of thought explanation synthesizing: For supervised fine-tuning with chain-of-thought (CoT), we aim to construct a training dataset comprising high-quality CoT. This enables the model not only to acquire the capability to decompile assembly code accurately but also to generate explicit, step-by-step rationales for the decompilation process, thereby enhancing the transparency and interpretability of the generated outputs. However, most large-parameter general models lack the domain-specific knowledge required to understand the bytecode, which leads to difficulties in synthesizing CoTs. To address this, we referenced the methodology in [60], using the source functions as hints for these large-parameter teacher models to generate decompilation CoT for the assembly functions. Specifically, we prompt the LLM to select critical instructions from the input TAC and ask it to identify the key instructions for decompilation. For each selected instruction, the LLM provides a brief explanation and maps it to the corresponding source code statement(s) that the instruction likely decompiles into. We also input the source functions, and emphasize in the prompt that these source functions should only be used to aid in understanding the bytecode and should not directly disclose the corresponding source code.

3.1.2 Test Case Synthesizing. Execution consistency is a critical metric for both the reinforcement learning (RL) training stage and the evaluation phase. To construct a dataset that enables program

execution and consistency checking, it is necessary to generate executable test cases that can effectively expose state discrepancies. Fuzzing [22, 36, 39] is adopted by DiSCo as a primary technique for assessing execution consistency during evaluation. While fuzzing is effective in uncovering inconsistencies by exploring diverse execution paths, it is not well-suited for RL training. This limitation stems from two main factors. First, fuzzing typically requires a large volume of test inputs, which incurs significant execution overhead and substantially increases training time. Second, fuzzing-generated inputs often lack sufficient discriminative power: a large number of fuzzing test cases may concentrate on a limited subset of execution paths. As a result, even if certain program branches are incorrectly implemented, the overall consistency score may still remain close to the maximum, thereby weakening the learning signal provided to the RL stage.

To mitigate these limitations, when using LLM to synthesize the test cases, we incorporate explicit requirements on branch coverage and precision into the prompt design. In particular, we encourage the LLM to generate test cases such that each one corresponds to a distinct program branch, ensuring that individual test cases provide comparable and informative training signals. Furthermore, since the execution of many branches depends on inter-procedural interactions, we explicitly instruct the model to consider the sequence of function calls in the prompt. This guidance facilitates the activation of deeper branches and improves the effectiveness of execution consistency checking for RL training. After generating test cases, we filter out those that cannot be executed. In summary, our prompt-guided test case synthesizing strategy provides efficient and informative execution signals for reinforcement learning. By emphasizing branch coverage, precision, and function-call sequence, it reduces redundancy and bias compared to fuzzing-based methods, resulting in more discriminative execution consistency feedback for effective RL training.

3.2 Chain-of-Thought Supervised Fine-Tuning

Our goal is to train a model that decompiles assembly into both correct high-level code and human-understandable explanations. We begin with supervised fine-tuning (SFT), using a synthetic dataset that pairs TAC and contextual information with the corresponding target source code and explanation. SFT provides the model with foundational decompilation competence and a coherent initialization for subsequent reinforcement learning; without it, the decompilation task is rarely solved correctly, causing RL rewards to collapse to sparse signals. This joint training of code and explanation enhances both accuracy and interpretability. Liu et al. [33] demonstrated that, when fine-tuning models for code generation, generating code before CoT reasoning significantly improves accuracy, whereas CoT preceding the code can sometimes degrade performance. Motivated by [33], we place the generated code before the CoT reasoning. Specifically, we combine the TAC function obtained in Section 3.1.1 with the corresponding contextual information and prepend appropriate task instructions to form the input for SFT. The supervision target consists of the source function corresponding to the TAC function, followed by a CoT explanation generated in Section 3.1.1.

3.3 Reinforcement Learning

SFT relies on surface-level supervision, encouraging syntactic similarity rather than semantic equivalence. As a result, the model may produce decompilation that appears valid but alters the original contract's runtime behavior. More critically, without execution-guided feedback or vulnerability-aware signals, the training objective is misaligned with our ultimate goal: generating Solidity code that is functionally equivalent to the original contract and faithfully preserves the vulnerabilities. To address these shortcomings, we introduce a subsequent reinforcement learning (RL) stage, wherein the model is optimized using execution-based and vulnerability-related rewards. Since the difficulty of function-level decompilation varies substantially, absolute reward values are not directly comparable across functions. Therefore, we adopt Group Relative Policy Optimization

(GRPO) [21], which updates the policy based on relative performance among sampled candidates and does not require a critic model. This design simplifies training while remaining aligned with our evaluation objectives.

GRPO's core innovation lies in its group-relative advantage estimation. Instead of using absolute reward values, GRPO normalizes each response's reward against the statistical distribution of rewards within its group (mean and standard deviation). This normalization effectively mitigates scale sensitivity and reduces variance across different batches. By measuring performance relative to peers within the same group, GRPO provides a stable learning signal even when reward magnitudes vary dramatically across different decompilation tasks. The objective function optimizes the policy model to maximize these group-relative advantages while constraining policy updates through a clipped objective similar to Proximal Policy Optimization (PPO) [41], incorporating importance sampling to enable off-policy learning and including a Kullback–Leibler (KL) divergence [28] penalty to prevent excessive deviation from a reference policy.

The design of our reward function is crucial since it directly shapes both the advantage estimation and policy updates. To effectively guide decompilation toward semantic fidelity and vulnerability preservation, we compose our reward from four complementary components:

- **Format Check (R_{format})** This component ensures that the model correctly structures its output using specific tags. The answer must be enclosed within *<answer></answer>* tags, while the explanation is placed within *<think></think>* tags. This format allows auditors to easily decompile the result and verify that the decompiled code corresponds to the bytecode. A compliant format yields a reward of $R_{\text{format}} = 1$.
- **Compilability Check ($R_{compile}$)** This is a fundamental prerequisite for code generation tasks. We use the Solidity compiler (solc) [4] to compile the result by replacing the source code function with the decompiled function. A successful compilation results in $R_{\text{compile}} = 1$, while a compilation failure results in $R_{\text{compile}} = 0$.
- **Execution Check ($R_{execution}$)** The execution check involves executing the decompiled code and deriving the reward by evaluating the consistency of storage states and return values across a set of test cases. Specifically, for each test case, we compare the operands of the **SSTORE** and **RETURN** instructions. If all operands are consistent in the execution for a given test case, we consider the decompiled function to have passed. If more than half of the operands are consistent, the function is deemed to have partially passed. The execution reward, $R_{\text{execution}}$, is calculated as the average of the individual test case rewards, as shown in Equation 1, where N represents the number of test cases (which is less than or equal to 5), and V_i is the reward for the i-th test case.

$$V_i = \begin{cases} 1.0, & \text{passed} \\ 0.5, & \text{partially passed} \\ 0, & \text{other} \end{cases}, \quad R_{\text{execution}} = \frac{\sum^{N} V_i}{N} \tag{1}$$

- **Vulnerability Check (R_{vul})** To ensure the decompilation faithfully preserves the security logic of the original bytecode, we perform a vulnerability consistency check. In the vulnerability consistency check, we examine whether the source function and the decompiled function exhibit the same defects or vulnerabilities. We employ Slither [16] to detect vulnerabilities in both the source and decompiled functions. Although Slither's vulnerability detection may produce false positives, we employ it as a relative consistency metric: because the same tool is applied to both the source and the decompiled code, any systematic bias affects both sides equally, allowing us to reliably compare whether the two versions trigger the same set of static-analysis alarms. The vulnerabilities detected in the source function are denoted as V_s, while those detected in

the decompiled function are denoted as V_d. The reward for vulnerability consistency between vulnerabilities in the source function and the decompiled function is measured by the Jaccard index $R_{\text{vul}} = |V_s \cap V_d| \,/\, |V_s \cup V_d|$.

The total reward R for each output is a weighted sum of these four scores, reflecting their relative importance:

$$R = \alpha_1 R_{format} + \alpha_2 R_{compile} + \alpha_3 R_{execution} + \alpha_4 R_{vul} \tag{2}$$

In designing the reward function, the weights $(\alpha_1, \alpha_2, \alpha_3, \alpha_4) = (0.1, 0.15, 0.6, 0.15)$ are assigned to balance the four evaluation components according to their relative importance for faithful decompilation. We prioritize semantic correctness: α_3 (dynamic execution consistency) receives the highest weight (0.6) because it most directly reflects whether the decompiled function preserves the original runtime behavior. The compilation success reward α_2 and the vulnerability preservation reward α_4 also significantly impact semantic faithfulness, and thus each receives a weight of 0.15. We assign the lowest weight to α_1 (formatting) (0.1) because syntactic style, while helpful for readability, is secondary to semantic accuracy and can be reasonably guided by supervised pretraining. This weight configuration was determined through multiple rounds of validation to effectively balance all objectives and prevent the model from generating overly short or incomplete code merely to pass compilation checks.

4 Experiment Setup

In this section, we formally present our research questions and detail the experimental results.

4.1 Research Questions

We investigate the following research questions to evaluate SmartDecompiler-R1.

RQ1: How effective is SmartDecompiler-R1 compared to other decompilation techniques?

RQ2: How effective is each component impact in SmartDecompiler-R1?

RQ3: Does our proposed Chain of Thought truly aid understanding for users or the model?

RQ4: How effective is SmartDecompiler-R1 in preserving vulnerabilities from bytecode?

4.2 Data Preparation

Our SFT training set is drawn from the training split of the smart contract dataset released by Storhaug et al [43], which contains 186,397 contracts. We employ **Vandal** [7] to generate bytecode-level functions and extract the corresponding source-level functions from Solidity code. For functions whose signatures are absent from our database, we filter them out to ensure an automatic and reliable matching between bytecode and source functions. After removing duplicates based on the hash values of source functions, we obtain a total of 33,610 *<bytecode, source>* function pairs for SFT. We then generate test cases for the smart contracts in the test set of [43]. After synthesizing the test cases, we execute them to filter out failed cases. Following the same deduplication and bytecode analysis procedure used for constructing the training data, we finally obtain 1,989 unique function pairs associated with a set of executable test suites. We split these 1,989 function pairs into two subsets: 1,200 for RL training and the remaining 789 for evaluation. We employ Qwen3-Coder-30B-A3B-Instruct [48] to synthesize explainable CoT rationales and test cases.

4.3 Experiment Setup

4.3.1 Metrics. To evaluate the quality of the decompiled code, we adopt four metrics: *BLEU-4*, *Compilability*, *Execution Consistency*, and *Vulnerability Consistency*. Specifically, we use BLEU-4 to measure lexical similarity, while the other three metrics capture functional correctness and

behavioral alignment. BLEU-4 scores are computed using **SacreBLEU** [38]. For remaining three metrics, we follow the same procedure used for reward computation, as described in Section 3.

4.3.2 Baselines. We compare SmartDecompiler-R1 against several representative baselines:

- **SmartHalo** [29]: a state-of-the-art method-level decompiler that leverages LLMs to improve variable recovery and infer contract-level attributes. Its outputs, however, are not designed to be directly compilable, as they typically include explanatory text and annotations with no fixed format. To enable a fair comparison, we manually extract the core function bodies from the raw LLM output, treating these extracted segments as the final decompiled code for assessment.
- **Dedaub+LLM**: a refinement-based baseline reported in DiSCo [44]. DiSCo is also a competitive smart contract decompilation approach. However, since neither the source code nor the original decompilation outputs of **DiSCo** are publicly available, we do not include it as a baseline in our evaluation. Instead, we follow the regeneration strategy baseline proposed by **DiSCo**, i.e., applying an LLM to refine the output of Dedaub (the web version of Elipmoc [20]). We instantiate this baseline with three representative LLMs: **GPT-5**, **Qwen3-Coder-480B**, and **DeepSeek-V3.2**.
- **DecompileLlama** [11]: a supervised learning-based approach that utilizes TAC–source function pairs along with contextual information for decompilation. We implement it using Llama-3.2-3B as the backbone. Following the original methodology, we supply state-variable analysis results as variable context and known public-function signatures as function context, using the prescribed input markers *[variables]*, *[functions]*, and *[TAC]* to separate the content. To strictly adhere to the original setting, we do not introduce any additional instructions.
- **DecompileQwen**: an adaptation of the DecompileLlama approach but with Qwen2.5-Coder-3B as the backbone to control for potential bias from base model capabilities. We train both DecompileLlama and DecompileQwen using full-parameter fine-tuning (as opposed to the LoRA-based [24] strategy in the original work) to ensure a fair comparison by eliminating performance discrepancies arising from different optimization configurations.

4.3.3 Implementation. All experiments are conducted on a machine equipped with four NVIDIA H100 GPUs (80GB HBM3 each). We adopt **Qwen2.5-Coder-3B-Instruct** [27] as the base model in all experiments[1]. During supervised fine-tuning, we train the model for 2 epochs with a learning rate of 1×10^{-5} and a batch size of 64. In the subsequent RL stage, we train the model for 1 epoch using the same learning rate and sample 8 candidate responses per prompt, with a maximum response length of 2048 tokens. We implement both SFT and RL training using the **ms-swift** [62] framework, together with **DeepSpeed** ZeRO-2 optimization.

5 Experiment Result

5.1 RQ1: Effectiveness

5.1.1 Experiment Setup. In this RQ, we evaluate the effectiveness of SmartDecompiler-R1 against the baseline methods described in Section 4.3.2 across our decompilation benchmark. We employ the same evaluation metrics (BLEU-4, compilability, execution consistency, and vulnerability consistency) and dataset splits for all methods.

5.1.2 Experiment Result. Table 1 reports the performance of SmartDecompiler-R1 compared with baseline methods on method-level decompilation. As shown in the table, SmartDecompiler-R1 consistently outperforms all competing approaches across the evaluated metrics, achieving 60.49%

[1]SmartDecompiler-R1 uses the instruction-tuned model so that RL can proceed without extra SFT, isolating each stage's contribution in ablation studies. DecompileQwen uses Qwen2.5-Coder-3B as the base model because it does not rely on instructions.

Table 1. Effectiveness Study: Performance Comparison Across Different Methods

Approach	BLEU-4	Compilability	Execution	Vul-Cons.	Average
SmartHalo	22.49	18.50	8.44	2.83	13.07
Dedaub+DeepSeek-V3.2	30.44	18.25	7.70	2.21	14.65
Dedaub+Qwen3-Coder-480B	34.57	21.80	11.39	2.44	17.55
Dedaub+GPT-5	40.09	21.92	12.31	2.94	19.32
DecompileLlama	54.58	36.88	22.92	20.96	33.84
DecompileQwen	60.07	38.66	23.34	20.64	35.68
SmartDecompiler-R1	**60.49**	**61.34**	**34.13**	**35.07**	**47.76**

on BLEU-4, 61.34% on compilability rate, 34.13% on execution consistency rate, and 47.76% on Vul-Consistency rate. Compared with the second-best method, DecompileQwen, SmartDecompiler-R1 achieves substantial improvements in compilability, execution consistency, and vulnerability consistency. These results indicate that reinforcement learning effectively guides the model to optimize toward preserving both vulnerable behaviors and execution consistency with the original code. **SmartHalo** demonstrates relatively poor performance in our evaluation, which can be attributed to two main reasons. First, its decompilation objective is not well aligned with ours. Although SmartHalo also operates at the method level, it typically generates a contract containing the target function as the decompilation result. To ensure compilability, it relies on LLMs to introduce additional state variables, which is problematic in our test suite: since the storage slot positions used to evaluate state variable consistency are fixed, such newly introduced variables lead to severe inconsistencies and consequently degrade performance. Second, SmartHalo focuses on recovering attributes of variables and contracts without explicitly constraining the model to preserve the original program logic. The lack of such constraints often results in excessive content generation and unintended logical deviations, further harming execution and vulnerability consistency.

Additionally, we observe that for methods based on LLMs, the vulnerability consistency scores are substantially lower than their performance on other metrics. This observation is consistent with the findings reported in DiSCo, which show that some post-decompilation inconsistencies arise because LLMs tend to automatically fix vulnerabilities. To further investigate this behavior, we analyze the outputs of LLM-based methods using Slither and find that these methods rarely reproduce vulnerabilities that can be detected by Slither. In other words, approaches employing LLMs tend to perform poorly in vulnerability preserving, largely because their original training objective emphasizes generating syntactically correct and vulnerability-free code rather than faithfully preserving buggy behaviors.

For the baselines that apply LLMs to optimize decompiled code, we observe that the capability of the underlying LLM has a noticeable impact on the quality of the resulting decompiled output. In the smart contract decompilation code regeneration task, GPT-5 outperforms DeepSeek-V3.2 and Qwen3-Coder-480B. Nevertheless, both LLM-based baselines perform worse than the learning-based approach. One key reason is that, although we explicitly instruct the LLMs to strictly follow the Dedaub-decompiled code, they still tend to introduce non-existent logic and generate statements that are not present in the original program. In addition, the regeneration process of LLMs is heavily influenced by the quality and characteristics of the Dedaub decompiled code. Since Dedaub aims to faithfully recover from low-level instructions through rule-based heuristic algorithms, its output may contain misleading representations, which in turn misguide the LLM during code regeneration. For instance, a simple array access to addrs may be decompiled as `keccak256(v6.data)` by Dedaub,

Table 2. Ablation Study: Impact of Different Training Components.

Model Variant	BLEU-4	Compilability	Execution	Vul-Cons.	Average
Base Model	19.95	19.90	4.33	8.16	13.08
Base+CA	21.70	15.96	4.54	6.23	12.11
Base+SFT	52.37	35.49	22.48	20.43	32.69
Base+RL	34.28	75.29	18.92	33.13	40.41
Base+SFT+RL	57.70	48.04	29.39	28.66	40.95
Base+CA+SFT	55.90	35.10	19.44	18.28	32.18
Base+CA+RL	29.03	**81.50**	20.03	**39.93**	42.62
Full Model	**60.49**	61.34	**34.13**	35.07	**47.76**

which subsequently leads the LLM to generate `keccak256(abi.encodePacked(addrs, value))`, introducing unnecessary and incorrect logic.

Summary: Overall, SmartDecompiler-R1 achieves the best performance on method-level decompilation across all evaluation metrics, significantly outperforming existing baselines. The results indicate that SmartDecompiler-R1 more effectively preserves execution semantics and vulnerable behaviors, whereas LLM-based approaches frequently introduce semantic deviations and fail to maintain vulnerability consistency.

5.1.3 Experiment Setup. In this RQ, we conduct an ablation study to evaluate the effectiveness of each component in our approach. Our method consists of three key components: Context Analysis (CA), Supervised Fine-Tuning (SFT), and Reinforcement Learning (RL). We systematically remove these components, either individually or in combination, and perform a total of eight experimental settings to comprehensively assess the contribution of each component.

5.2 RQ2: Ablation Study

5.2.1 Experiment Result. Table 2 shows that removing any component degrades SmartDecompiler-R1, demonstrating the contribution of each component. Unlike the effectiveness experiments, where metrics maintain a partial positive correlation, the ablation study exhibits a clear metric imbalance. Since metric importance is unequal, mean-based comparison is coarse; thus, under severe imbalance, we emphasize BLEU-4 and execution consistency as indicators of syntactic fidelity and runtime semantic equivalence. A decompilation is considered successful if either of these two metrics reaches 100%, whereas compilability rate and vulnerability consistency rate alone are insufficient.

We first examine the role of the Context Analysis stage, which supplements SmartDecompiler-R1 with auxiliary semantic signals that are difficult to infer solely from the input TAC function. As shown in Table 2, removing this component from the full model leads to a clear performance degradation, particularly in terms of code compilability. This suggests that the additional structural information introduced by context analysis, such as storage slot resolution and function call identification, plays an important supporting role in recovering public state variables and function interfaces, thereby improving the robustness of the generated Solidity code.

Next, we analyze the impact of the SFT stage. From Table 2, we observe that SFT has a clear positive effect on SmartDecompiler-R1, with the most pronounced improvement appearing on the BLEU-4 score. Removing the SFT stage leads to a significant degradation in BLEU-4, indicating that SFT plays a crucial role in learning the mapping from TAC to high-quality Solidity code. Although not explicitly reflected in the quantitative metrics, the SFT stage also enables SmartDecompiler-R1 to generate meaningful explanations during the decompilation process by leveraging synthetic CoT

data. When evaluating models trained without SFT on samples containing a synthetic CoT, we find that they fail to produce detailed and accurate reasoning traces. This suggests that SFT is essential for aligning the model with the intended decompilation and explanation behavior. Overall, the SFT stage provides SmartDecompiler-R1 with foundational capabilities in both decompilation and explanation, thereby laying the groundwork for subsequent RL stage to further refine performance.

From Table 2, we can clearly observe that the RL stage consistently enhances SmartDecompiler-R1 's performance across all evaluation metrics, similar to the improvements brought by the SFT stage. In particular, for the full model, RL leads to significant gains in compilability, execution consistency, and vulnerability consistency, indicating that RL effectively leverages context analysis signals and further refines the capabilities initially developed during SFT. However, we also find that applying RL without SFT results in an imbalanced model behavior. Specifically, the model tends to generate code with high compilability and vulnerability consistency, but shows less significant improvements on BLEU-4 and execution consistency. As discussed earlier, BLEU-4 and execution consistency are more crucial indicators of overall code quality and functional correctness. Therefore, using RL in isolation to improve only compilability and vulnerability recovery ability is not a desirable strategy. This imbalance likely stems from the model's limited capacity to generate high-quality code when trained solely with RL. In such cases, the model is prone to reward hacking, where it exploits shortcuts in the reward function to maximize its score without genuinely improving code quality. Specifically, it gravitates toward easier-to-obtain rewards, such as those associated with compilability (by minimizing code length) and vulnerability preservation (by targeting simple, surface-level vulnerabilities). Nevertheless, the ablation results confirm that RL remains a valuable and effective component when used in conjunction with SFT and CA. It serves as a powerful refinement step that enhances the robustness and reliability of the code decompiled by the model.

Summary: Overall, the ablation study demonstrates that Context Analysis, Supervised Fine-Tuning, and Reinforcement Learning are all indispensable components of SmartDecompiler-R1. Both SFT and RL substantially enhance the model's decompilation capability. Moreover, RL further improves the compilation success rate, execution consistency, and vulnerability consistency, while also enabling the model to more effectively leverage the contextual information provided by context analysis.

5.3 RQ3: Effectiveness of Explanation

5.3.1 Experimental Setup. In this RQ, to investigate the effectiveness of the explanations generated by SmartDecompiler-R1, we adopt a complementary evaluation paradigm. Although these explanations are designed to assist human auditors, large-scale manual assessment is impractical due to the limited availability of experienced smart-contract bytecode auditors. Given that existing work [49, 63] has demonstrated the feasibility of using LLMs as automated evaluators for code, we leverage LLMs to approximate the benefits brought by the explanations.

We evaluate the effectiveness of explanations generated by SmartDecompiler-R1 by testing whether they can effectively guide LLMs to produce higher-quality decompiled code. Specifically, we prompt the same LLM to decompile a given TAC function under three conditions: ① without any explanation. ② with a CoT explanation generated by Deepseek-V3.2 during its reasoning process. ③ with the explanation produced by SmartDecompiler-R1 provided as the CoT guidance. We then compare the quality of the resulting decompiled functions across these three settings. An improvement in decompilation quality when explanations are provided indicates that the explanations are informative and effective. To ensure robustness, we employ three instruction-tuned code LLMs—Qwen2.5-Coder-7B, CodeLlama-7B, and DeepSeek-Coder-6.7B—as evaluators

Table 3. Chain-of-Thought Study: Impact of Explanation Generation

Model	CoT Hint	BLEU-4	Compil.	Execution	Vul-Cons.	Average
Qwen2.5-Coder-7B	✗	17.43	20.66	3.29	2.33	10.93
	DS	16.70	18.76	2.84	3.68	10.50
	Ours	34.32	26.49	6.74	13.46	20.25
CodeLlama-7B	✗	4.82	10.52	1.89	3.08	5.08
	DS	7.87	15.33	3.61	4.32	7.78
	Ours	13.61	25.73	9.22	12.82	15.35
DeepSeek-Coder-6.7B	✗	17.53	30.29	3.52	9.33	15.17
	DS	13.10	20.41	3.90	4.74	10.54
	Ours	39.34	34.09	9.82	17.98	25.31

and measure performance using multiple complementary metrics, including BLEU-4, compilability, execution consistency, and vulnerability consistency.

5.3.2 Experiment Result. As shown in Table 3, providing LLM evaluators with explanations generated by SmartDecompiler-R1 as CoT guidance leads to substantial and consistent improvements in decompilation performance. Across all evaluated models and all metrics—including BLEU-4, compilability, execution consistency, and vulnerability consistency—models equipped with explanation guidance significantly outperform their counterparts without explanations. On average, explanation-guided settings achieve nearly double the overall score compared to the no-explanation baseline. These results demonstrate that the explanations produced by SmartDecompiler-R1 effectively capture semantic and structural information that facilitates the decompilation process.

Notably, the CoT explanations generated by DeepSeek-V3.2 do not consistently improve decompilation quality. For Qwen2.5-Coder-7B and Deepseek-Coder-6.7B, providing DeepSeek-V3.2's CoT can even degrade performance. This contrast underscores that the explanations produced by SmartDecompiler-R1 are more targeted and informative than generic reasoning-based CoTs from DeepSeek-V3.2. Because the evaluated LLMs serve as proxies for human auditors, these results indirectly suggest that SmartDecompiler-R1 's explanations could also help human analysts better understand low-level bytecode and reason about its high-level semantics.

Summary: Using explanations generated by SmartDecompiler-R1 as CoT guidance consistently improves decompilation performance across all evaluators and metrics, indirectly indicating that our targeted explanation helps auditors better understand low-level instructions.

5.4 RQ4: Vulnerabilities Preservation

5.4.1 Experiment Setup. In this RQ, we investigate whether SmartDecompiler-R1 can faithfully preserve vulnerabilities that exist in the original source functions after decompilation. For this purpose, we compile a curated dataset [32] with 15 access-control (AC) vulnerabilities sourced from publicly reported CVEs. Our evaluation focuses specifically on AC vulnerabilities, as they constitute a prevalent and critical category of logical flaws in smart contracts, are well-documented in prior research [18, 32]. Additionally, AC vulnerabilities are relatively straightforward to identify manually. This allows us to examine whether such security-critical semantics are retained in the decompiled outputs.

For each vulnerable bytecode function, we decompile it using SmartDecompiler-R1 and three LLM-enhanced baselines: Dedaub combined with GPT-5, Qwen3-Coder-480B, or DeepSeek-V3.2. We select these LLM-enhanced baselines (rather than end-to-end models like DecompileLlama) to

Table 4. Vulnerability Preservation

Approach	Successful	Partially Successful	Failed	Auto-Fix
Dedaub+GPT-5	10	1	1	3
Dedaub+Qwen3-Coder-480B	9	0	1	5
Dedaub+Deepseek-V3.2	5	2	1	7
SmartDecompiler-R1	11	2	2	0

explicitly test whether these powerful LLMs automatically fix vulnerabilities during decompilation. Then, we manually inspect the decompiled code to determine whether the original vulnerability is preserved. We classify the decompilation results into four categories: ① **Successful:** The decompiled function preserves the same vulnerability as the original source code and correctly recovers core logic. ② **Partially successful**: The decompiled function preserves the vulnerability, but fails to recover some non-security-related logic. ③ **Failed**: The decompiled function is semantically incorrect or incomprehensible, making vulnerability assessment impossible. ④ **Auto-Fix**: The decompiled function is semantically coherent but unintentionally eliminates the original vulnerability.

5.4.2 Experiment Result. Table 4 presents that SmartDecompiler-R1 demonstrates a strong capability for preserving security vulnerabilities during decompilation. Specifically, SmartDecompiler-R1 achieved an Auto-Fix score of 0, meaning it did not automatically "repair" any vulnerability in any of the cases. It successfully retained the original vulnerability in 11 cases, with 2 cases partially successful and 2 cases failing to preserve the vulnerability. In contrast, all three LLM-enhanced baselines exhibited some degree of unintended vulnerability correction. Dedaub+Deepseek-V3.2 showed the highest auto-fix tendency, modifying 7 cases. Dedaub+Qwen3-Coder3-480B auto-fixed 5 cases, while Dedaub+GPT-5 auto-fixed 3 cases. These findings highlight that SmartDecompiler-R1 is more faithful to the original vulnerable semantics and does not introduce unwarranted "security fixes" that would mislead subsequent vulnerability analysis.

To further illustrate the effectiveness of SmartDecompiler-R1, we present three representative cases, as shown in Figure 4. In the *partially successful* case (CVE-2018-19834), the original function lacks access control, allowing any caller to become the owner. SmartDecompiler-R1 correctly retains this vulnerability, but incorrectly recovers the constant `INITIAL_SUPPLY`, leading to a mismatch in non-security-related logic. In the *failed* case (CVE-2020-17753), the original vulnerable condition `require(tx.origin == owner)` is not accurately reconstructed; instead, the decompiled code generates `require(msg.sender == tx.origin)`, which alters the security semantics and fails to reflect the original vulnerability. In the *successful* case (CVE-2019-15078), both the logic and the access-control flaw are fully preserved, demonstrating SmartDecompiler-R1 's ability to faithfully retain security-critical semantics. These examples highlight that while SmartDecompiler-R1 generally preserves vulnerabilities, its success depends on accurately reconstructing both control-flow and data-flow elements that embody the security flaw.

Summary: SmartDecompiler-R1 consistently preserves original vulnerabilities without auto-fixing, demonstrating high semantic faithfulness while LLM-enhanced baselines frequently introduce unintended vulnerability corrections.

6 Discussion

In this section, we discuss the quality of the synthetic dataset and the threats to validity in our study.

```
cve-2018-19834
//source code
uint256 public constant INITIAL_SUPPLY = 10000000;
function quaker(address wallet) public {
  owner = msg.sender;
  ownerWallet=wallet;
  totalSupply = INITIAL_SUPPLY * 10 ** 18;
  tokenBalances[wallet] = INITIAL_SUPPLY * 10 ** 18;
 }

//decompiled code
function quaker(address _quaker) public {
  owner = msg.sender;
  quakerAddress = _quaker;
  totalSupply = 1000000000 * (1 ether);
  balances[_quaker] = 1000000000 * (1 ether);
 }
```

(a) Partially Successful Case

```
cve-2020-17753
//source code
function addMeByRC() public {
  require(tx.origin == owner);
  rc[msg.sender] = true;
  emit NewRC(msg.sender);
 }


//decompiled code
function addMeByRC() public {
  require(msg.sender == tx.origin);
  rc[msg.sender] = true;
  AddMeByRC(msg.sender);
 }
```

(b) Failed Case

```
cve-2019-15078
//source code
function XBornID() public {
  owner = msg.sender;
  balances[owner] = totalDistributed;
 }


//decompiled code
function XBornID() public {
  owner = msg.sender;
  balances[owner] = totalDistributed;
 }
```

(c) Successful Case

Fig. 4. Vulnerability Preservation Cases

6.1 Synthetic Dataset Quality

To evaluate the quality of our synthetic dataset, we examined branch coverage on a random sample of 30 source functions (each with at least one non-assertion branch) and their test cases. Manual inspection revealed 94 branches in total, of which 70 were covered. Of the 24 uncovered branches, 12 depended on blockchain-specific conditions (e.g., block timestamp or number) that could not be satisfied in our test environment, and 2 required repeated function calls beyond our execution limit. This indicates that our dataset achieves solid practical coverage and is capable of reliably detecting runtime behavioral inconsistencies.

We also observed that the synthetic dataset may not fully exercise certain assertion statements, such as **require**. The limited number of test cases can occasionally make it difficult to reliably assess the correctness of **require** statement implementations, particularly in functions that contain multiple such assertions. As a result, our runtime consistency metric, which is based primarily on test case behavior, might not always completely reflect the effectiveness of program recovery on **require** statements.

6.2 Threats to Validity

6.2.1 Internal Validity. First, our function-level decompilation approach may introduce inconsistencies when reconstructing full contracts, due to limited global context. As Liu et al. [34] underscore, evaluation beyond the function level is critical for assessing the practical utility of decompilers, since many real-world analyses depend on contract-level semantics. While our current scoping decision was motivated by the need to isolate fine-grained translation accuracy, we acknowledge this limitation and plan to explore hierarchical or modular approaches for contract-level reconstruction in future work. Second, the model's performance depends on the quality of the input TAC produced by front-end tools, which can be affected by heavily optimized or shared-control-flow bytecode. We mitigate this by building on a widely-used analysis framework (Vandal). Finally, the weights in the RL reward function are set empirically, which could bias optimization. These weights were, however, calibrated through iterative validation on a held-out set to balance core objectives. Moreover, by employing GRPO, which relies on relative rewards within groups, the policy update becomes less sensitive to absolute reward magnitudes than critic-based methods.

6.2.2 External Validity. First, our model is trained and evaluated on Solidity datasets, and may not generalize to heavily obfuscated code, rare patterns, or other languages (e.g., Vyper). To improve generalization, we use a large and diverse public dataset and encourage semantic learning through execution-based RL rewards. Second, the usefulness of generated explanations is assessed via LLM-based proxy tasks, not directly by human auditors. However, LLM-based evaluation serves as a scalable preliminary indicator of explanatory utility. The consistent correlation between

explanation guidance and enhanced decompilation quality across multiple LLMs provides indirect but supporting evidence.

7 Related work

7.1 Reinforcement Learning in Software Engineering

Recently, reinforcement learning (RL) has attracted growing attention in software engineering. Prior studies have explored RL for a variety of tasks, including bug localization, software testing, and code completion. Chakraborty et al. [8] leverage RL to improve the effectiveness of bug localization. In software testing, Eom [13] applies RL to guide fuzzing toward generating inputs that achieve higher coverage. RL has also been adopted for repository-level code completion to better retrieve relevant context from large codebases [53]. In the smart contract domain, **Smart-LLaMA-DPO** [57] employs Direct Preference Optimization (DPO) [40] to train an explainable vulnerability detector, while **SmartCoder-R1** [58] uses GRPO-based RL to develop a security-aware smart contract generator.

7.2 LLM for Decompilation

The strong capability of large language models (LLMs) on programming-related tasks has recently motivated their adoption in decompilation pipelines. DecLLM [55] uses an LLM as a post-processing module to refine decompiled C code by repairing static inconsistencies and mitigating runtime failures, while DeGPT [25] improves readability via a multi-role prompting scheme for structure simplification and code rewriting. To alleviate the mismatch between LLMs and low-level bytecode, DiSCo [44] converts bytecode into higher-level natural-language representations to facilitate LLM reasoning. Beyond using off-the-shelf LLMs, LLM4Decompile [47] performs self-supervised fine-tuning to train an end-to-end model for decompiling C assembly, and further adopts a similar approach to optimize decompiled outputs, improving reconstruction accuracy.

8 Conclusion

In this paper, we presented SmartDecompiler-R1, a novel framework for decompiling EVM bytecode into high-level source code. By leveraging reinforcement learning, SmartDecompiler-R1 significantly improves the performance of learning-based decompilation approaches. Extensive experimental evaluation demonstrates that SmartDecompiler-R1 outperforms existing decompilation methods in terms of consistency, and offers a new perspective to assist auditors in understanding bytecode by generating human-readable explanations. Furthermore, our results show that SmartDecompiler-R1 is effective in preserving vulnerabilities from EVM bytecode.

Acknowledgement

This work is supported by Zhejiang Provincial Science Foundation of China under Grant No. LQK26F020002, the National Science Foundation of China (No. 62372398, No. 62572322, No. 72342025), and the Fundamental Research Funds for the Central Universities (No. 226-2025-00067).

Data Availability

The experimental results, along with all data, code, and checkpoints required to reproduce the experiments reported in this paper, are available at: https://github.com/mmmmmyl/SmartDecompiler-R1.git.

References

[1] 2019. Panoramix. https://github.com/eveem-org/panoramix. Accessed: 2025-12-30.
[2] 2019. porosity. https://github.com/msuiche/porosity. Accessed: 2025-12-30.
[3] 2020. octopus. https://github.com/FuzzingLabs/octopus. Accessed: 2025-12-30.

[4] 2025. Solidity. https://github.com/argotorg/solidity. Accessed: 2025-12-30.

[5] Elvira Albert, Pablo Gordillo, Benjamin Livshits, Albert Rubio, and Ilya Sergey. 2018. EthIR: A Framework for High-Level Analysis of Ethereum Bytecode. In *Automated Technology for Verification and Analysis*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer International Publishing, Cham, 513–520.

[6] blockscout. 2025. *Blockchain explorer for Ethereum based network and a tool for inspecting and analyzing EVM based blockchains.* https://www.blockscout.com/ Accessed: Dec., 2025.

[7] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, Francois Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. 2018. Vandal: A Scalable Security Analysis Framework for Smart Contracts. arXiv:1809.03981 [cs.PL] https://arxiv.org/abs/1809.03981

[8] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. 2024. RLocator: Reinforcement Learning for Bug Localization. *IEEE Trans. Softw. Eng.* 50, 10 (Oct. 2024), 2695–2708. doi:10.1109/TSE.2024.3452595

[9] Jiachi Chen, Yiming Shen, Jiashuo Zhang, Zihao Li, John Grundy, Zhenzhe Shao, Yanlin Wang, Jiashui Wang, Ting Chen, and Zibin Zheng. 2025. FORGE: An LLM-driven Framework for Large-Scale Smart Contract Vulnerability Dataset Construction. arXiv:2506.18795 [cs.CR] https://arxiv.org/abs/2506.18795

[10] Jiachi Chen, Xin Xia, David Lo, John Grundy, Xiapu Luo, and Ting Chen. 2022. DefectChecker: Automated Smart Contract Defect Detection by Analyzing EVM Bytecode. *IEEE Transactions on Software Engineering* 48, 7 (2022), 2189–2207. doi:10.1109/TSE.2021.3054928

[11] Isaac David, Liyi Zhou, Dawn Song, Arthur Gervais, and Kaihua Qin. 2025. Decompiling Smart Contracts with a Large Language Model. arXiv:2506.19624 [cs.CR] https://arxiv.org/abs/2506.19624

[12] Dedaub. 2025. Dedaub: Smart Contract Security and Blockchain Monitoring. https://dedaub.com/ Accessed: 2025-12-30.

[13] Jueon Eom, Seyeon Jeong, and Taekyoung Kwon. 2024. Fuzzing JavaScript Interpreters with Coverage-Guided Reinforcement Learning for LLM-Based Mutation. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 1656–1668. doi:10.1145/3650212.3680389

[14] Ethereum. 2025. *Solidity Documentation.* https://docs.soliditylang.org Accessed: Dec., 2025.

[15] etherscan. 2025. *The Ethereum Blockchain Explorer.* https://etherscan.io/ Accessed: Dec., 2025.

[16] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: A Static Analysis Framework for Smart Contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. 8–15. doi:10.1109/WETSEB.2019.00008

[17] Zhipeng Gao, Vinoj Jayasundara, LingXiao Jiang, Xin Xia, David Lo, and John Grundy. 2019. SmartEmbed: A Tool for Clone and Bug Detection in Smart Contracts through Structural Code Embedding. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 394–397. doi:10.1109/ICSME.2019.00067

[18] Asem Ghaleb, Julia Rubin, and Karthik Pattabiraman. 2023. AChecker: Statically Detecting Smart Contract Access Control Vulnerabilities. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 945–956. doi:10.1109/ICSE48619.2023.00087

[19] Neville Grech, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2019. Gigahorse: Thorough, Declarative Decompilation of Smart Contracts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1176–1186. doi:10.1109/ICSE.2019.00120

[20] Neville Grech, Sifis Lagouvardos, Ilias Tsatiris, and Yannis Smaragdakis. 2022. Elipmoc: advanced decompilation of Ethereum smart contracts. *Proc. ACM Program. Lang.* 6, OOPSLA1, Article 77 (April 2022), 27 pages.

[21] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, and et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948 [cs.CL] https://arxiv.org/abs/2501.12948

[22] Jingxuan He, Mislav Balunović, Nodar Ambroladze, Petar Tsankov, and Martin Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 531–548. doi:10.1145/3319535.3363230

[23] Jiahao He, Shuangyin Li, Xinming Wang, Shing-Chi Cheung, Gansen Zhao, and Jinji Yang. 2023. Neural-FEBI: Accurate function identification in Ethereum Virtual Machine bytecode. *Journal of Systems and Software* 199 (2023), 111627. doi:10.1016/j.jss.2023.111627

[24] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL] https://arxiv.org/abs/2106.09685

[25] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, Vol. 267622140.

[26] Xing Hu, Zhipeng Gao, Xin Xia, David Lo, and Xiaohu Yang. 2021. Automating User Notice Generation for Smart Contract Functions. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 5–17. doi:10.1109/ASE51524.2021.9678552

[27] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (2024).

[28] Solomon Kullback. 1951. Kullback-leibler divergence. *Tech. Rep.* (1951).

[29] Zeqin Liao, Yuhong Nan, Zixu Gao, Henglong Liang, Sicheng Hao, Peifan Ren, and Zibin Zheng. 2025. Augmenting Smart Contract Decompiler Output through Fine-grained Dependency Analysis and LLM-facilitated Semantic Recovery. *IEEE Transactions on Software Engineering* (2025), 1–17. doi:10.1109/TSE.2025.3623325

[30] Zeqin Liao, Yuhong Nan, Henglong Liang, Sicheng Hao, Juan Zhai, Jiajing Wu, and Zibin Zheng. 2024. SmartAxe: Detecting Cross-Chain Vulnerabilities in Bridge Smart Contracts via Fine-Grained Static Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 12 (July 2024), 22 pages. doi:10.1145/3643738

[31] Zeqin Liao, Zibin Zheng, Xiao Chen, and Yuhong Nan. 2022. SmartDagger: a bytecode-based static analysis approach for detecting cross-contract vulnerability. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 752–764. doi:10.1145/3533767.3534222

[32] Huarui Lin, Zhipeng Gao, Jiachi Chen, Xiang Chen, Xiaohu Yang, and Lingfeng Bao. 2025. ACTAINT: Agent-Based Taint Analysis for Access Control Vulnerabilities in Smart Contracts. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2554–2566. doi:10.1109/ASE63991.2025.00210

[33] Renbiao Liu, Anqi Li, Chaoding Yang, Hui Sun, and Ming Li. 2025. Revisiting Chain-of-Thought in Code Generation: Do Language Models Need to Learn Reasoning before Coding?. In *Forty-second International Conference on Machine Learning, ICML 2025, Vancouver, BC, Canada, July 13-19, 2025*. OpenReview.net. https://openreview.net/forum?id=wSZeQoJ1Vk

[34] Xia Liu, Baojian Hua, Yang Wang, and Zhizhong Pan. 2023. An Empirical Study of Smart Contract Decompilers. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1–12. doi:10.1109/SANER56733.2023.00011

[35] Pengxiang Ma, Ningyu He, Yuhua Huang, Haoyu Wang, and Xiapu Luo. 2023. Abusing the Ethereum Smart Contract Verification Services for Fun and Profit. arXiv:2307.00549 [cs.CR] https://arxiv.org/abs/2307.00549

[36] Tai D. Nguyen, Long H. Pham, Jun Sun, Yun Lin, and Quang Tran Minh. 2020. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 778–788. doi:10.1145/3377811.3380334

[37] Minh V. T. Pham, Huy N. Phan, Hoang N. Phan, Cuong Le Chi, Tien N. Nguyen, and Nghi D. Q. Bui. 2025. SWE-Synth: Synthesizing Verifiable Bug-Fix Data to Enable Large Language Models in Resolving Real-World Bugs. arXiv:2504.14757 [cs.SE] https://arxiv.org/abs/2504.14757

[38] Matt Post. 2018. A Call for Clarity in Reporting BLEU Scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*. Association for Computational Linguistics, Belgium, Brussels, 186–191. https://www.aclweb.org/anthology/W18-6319

[39] Peng Qian, Hanjie Wu, Zeren Du, Turan Vural, Dazhong Rong, Zheng Cao, Lun Zhang, Yanbin Wang, Jianhai Chen, and Qinming He. 2024. MuFuzz: Sequence-Aware Mutation and Seed Mask Guidance for Blockchain Smart Contract Fuzzing. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 1972–1985. doi:10.1109/ICDE60146.2024.00158

[40] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *Advances in neural information processing systems* 36 (2023), 53728–53741.

[41] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347 [cs.LG] https://arxiv.org/abs/1707.06347

[42] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. doi:10.18653/v1/p16-1162

[43] André Storhaug, Jingyue Li, and Tianyuan Hu. 2023. Efficient avoidance of vulnerabilities in auto-completed smart contract code using vulnerability-constrained decoding. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 683–693.

[44] Xing Su, Hanzhong Liang, Hao Wu, Ben Niu, Fengyuan Xu, and Sheng Zhong. 2025. DiSCo: Towards Decompiling EVM Bytecode to Source Code using Large Language Models. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE103 (June 2025), 24 pages. doi:10.1145/3729373

[45] Yu Sun, Lingfeng Bao, and Xiaohu Yang. 2025. FIRE: Smart Contract Bytecode Function Identification via Graph-Refined Hybrid Feature Encoding. In *Proceedings of the 16th International Conference on Internetware (Internetware '25)*. Association for Computing Machinery, New York, NY, USA, 378–388. doi:10.1145/3755881.3755883

[46] Nick Szabo. 1997. Formalizing and securing relationships on public networks. *First monday* (1997).

[47] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 3473–3487. doi:10.18653/v1/2024.emnlp-main.203

[48] Qwen Team. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] https://arxiv.org/abs/2505.09388

[49] Weixi Tong and Tianyi Zhang. 2024. CodeJudge: Evaluating Code Generation with Large Language Models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen (Eds.). Association for Computational Linguistics, Miami, Florida, USA, 20032–20051. doi:10.18653/v1/2024.emnlp-main.1118

[50] Vyperlang. 2025. *Vyper Documentation.* https://docs.vyperlang.org/en/stable/ Accessed: Dec., 2025.

[51] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Vienna, Austria) *(ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 503–515. doi:10.1145/3650212.3652145

[52] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Online and Punta Cana, Dominican Republic, 8696–8708. doi:10.18653/v1/2021.emnlp-main.685

[53] Yanlin Wang, Yanli Wang, Daya Guo, Jiachi Chen, Ruikai Zhang, Yuchi Ma, and Zibin Zheng. 2025. *RLCoder: Reinforcement Learning for Repository-Level Code Completion.* IEEE Press, 1140–1152. https://doi.org/10.1109/ICSE55347.2025.00014

[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[55] Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA081 (June 2025), 24 pages. doi:10.1145/3728958

[56] J Xiang, Z Gao, and L Bao. 2025. Automating comment generation for smart contract from bytecode. *ACM Transactions on Software Engineering and Methodology* (2025).

[57] Lei Yu, Zhirong Huang, Hang Yuan, Shiqi Cheng, Li Yang, Fengjun Zhang, Chenjie Shen, Jiajia Ma, Jingyuan Zhang, Junyi Lu, and Chun Zuo. 2025. Smart-LLaMA-DPO: Reinforced Large Language Model for Explainable Smart Contract Vulnerability Detection. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA009 (June 2025), 24 pages. doi:10.1145/3728878

[58] Lei Yu, Jingyuan Zhang, Xin Wang, Jiajia Ma, Li Yang, and Fengjun Zhang. 2025. Towards Secure and Explainable Smart Contract Generation with Security-Aware Group Relative Policy Optimization. arXiv:2509.09942 [cs.CR] https://arxiv.org/abs/2509.09942

[59] Xiao Liang Yu, Omar Al-Bataineh, David Lo, and Abhik Roychoudhury. 2020. Smart Contract Repair. *ACM Trans. Softw. Eng. Methodol.* 29, 4, Article 27 (Sept. 2020), 32 pages.

[60] Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. STaR: Bootstrapping Reasoning With Reasoning. arXiv:2203.14465 [cs.LG] https://arxiv.org/abs/2203.14465

[61] Bolun Zhang, Zeyu Gao, Hao Wang, Yuxin Cui, Siliang Qin, Chao Zhang, Kai Chen, and Beibei Zhao. 2025. BinQuery: A Novel Framework for Natural Language-Based Binary Code Retrieval. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA052 (June 2025), 23 pages. doi:10.1145/3728927

[62] Yuze Zhao, Jintao Huang, Jinghan Hu, Xingjun Wang, Yunlin Mao, Daoze Zhang, Zeyinzi Jiang, Zhikai Wu, Baole Ai, Ang Wang, Wenmeng Zhou, and Yingda Chen. 2024. SWIFT:A Scalable lightWeight Infrastructure for Fine-Tuning. arXiv:2408.05517 [cs.CL] https://arxiv.org/abs/2408.05517

[63] Xin Zhou, Bowen Xu, Kisub Kim, DongGyun Han, Hung Huu Nguyen, Thanh Le-Cong, Junda He, Bach Le, and David Lo. 2024. Leveraging Large Language Model for Automatic Patch Correctness Assessment. *IEEE Transactions on Software Engineering* 50, 11 (2024), 2865–2883. doi:10.1109/TSE.2024.3452252

[64] Yi Zhou, Deepak Kumar, Surya Bakshi, Joshua Mason, Andrew Miller, and Michael Bailey. 2018. Erays: reverse engineering ethereum's opaque smart contracts. In *Proceedings of the 27th USENIX Conference on Security Symposium* (Baltimore, MD, USA) *(SEC'18)*. USENIX Association, USA, 1371–1385.

Received 2026-01-30; accepted 2026-04-16