SmartCoder-R1: Towards Secure and Explainable Smart Contract Generation with Security-Aware Group Relative Policy Optimization

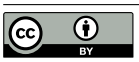
LEI YU*, Institute of Software, Chinese Academy of Sciences, China
JINGYUAN ZHANG*, Institute of Software, Chinese Academy of Sciences, China
XIN WANG*, Institute of Software, Chinese Academy of Sciences, China
LI YANG†, Institute of Software, Chinese Academy of Sciences, China
FENGJUN ZHANG†, Institute of Software, Chinese Academy of Sciences, China
JIAJIA MA, Institute of Software, Chinese Academy of Sciences, China

Smart contracts automate the management of high-value assets, where vulnerabilities can lead to catastrophic financial losses. In the task of automated smart contract generation using Large Language Models (LLMs), this challenge is amplified by two interconnected failures: first, they operate as unauditable "black boxes" by failing to produce a transparent reasoning process, and second, as a consequence, they generate code riddled with critical security vulnerabilities. To address both issues, we propose SmartCoder-R1 based on Qwen2.5-Coder-7B, a novel framework for secure and explainable smart contract generation. It begins with Continual Pre-training (CPT) to specialize the base model on the nuances of smart contract code. To construct the data for subsequent stages, we first prompt the DeepSeek model to generate reasoning-and-code samples from verified on-chain contracts, followed by a rigorous validation process where each sample is manually reviewed by security experts for compilability, functionality, security, and reasoning completeness. Based on this, we then apply Long Chain-of-Thought Supervised Fine-Tuning (L-CoT SFT) on 7,998 of these expert-validated samples to train the model to emulate human security analysis. Finally, to directly mitigate vulnerabilities, we employ Security-Aware Group Relative Policy Optimization (S-GRPO), a reinforcement learning phase that refines the generation policy using 1,691 samples by optimizing a weighted reward signal for compilation success, security compliance, and format correctness. Evaluated against 18 state-of-the-art baselines on a challenging benchmark of 756 real-world functions from 289 deployed contracts, SmartCoder-R1 establishes a new state of the art by achieving top performance across five key metrics: a ComPass of 87.70%, a VulRate of 8.60%, a SafeAval of 80.16%, a FuncRate of 53.84%, and a FullRate of 50.53%. This FullRate marks a 45.79% relative improvement over the strongest baseline, DeepSeek-R1. Crucially, its generated reasoning also excels in human evaluations, achieving high-quality ratings for Functionality (82.7%), Security (85.3%), and Clarity (90.7%).

CCS Concepts: • **Security and privacy → Software security engineering**; • **Software and its engineering** → *Domain specific languages.*

*Affiliated with University of Chinese Academy of Sciences, Beijing, China.
†Li Yang and Fengjun Zhang are the corresponding authors.

Authors' Contact Information: Lei Yu, Institute of Software, Chinese Academy of Sciences, Beijing, China, yulei2022@iscas.ac.cn; Jingyuan Zhang, Institute of Software, Chinese Academy of Sciences, Beijing, China, zhangjingyuan2023@iscas.ac.cn; Xin Wang, Institute of Software, Chinese Academy of Sciences, Beijing, China, wangxin@iscas.ac.cn; Li Yang, Institute of Software, Chinese Academy of Sciences, Beijing, China, yangli2017@iscas.ac.cn; Fengjun Zhang, Institute of Software, Chinese Academy of Sciences, Beijing, China, fengjun@iscas.ac.cn; Jiajia Ma, Institute of Software, Chinese Academy of Sciences, Beijing, China, majiajia@iscas.ac.cn.



ACM 2994-970X/2026/7-ARTFSE097
https://doi.org/10.1145/3808104

Additional Key Words and Phrases: Smart Contract, Code Generation, Large Language Models, Security-Aware Group Relative Policy Optimization

ACM Reference Format:
Lei Yu, Jingyuan Zhang, Xin Wang, Li Yang, Fengjun Zhang, and Jiajia Ma. 2026. SmartCoder-R1: Towards Secure and Explainable Smart Contract Generation with Security-Aware Group Relative Policy Optimization. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE097 (July 2026), 24 pages. https://doi.org/10.1145/3808104

1 Introduction

Blockchain technology has been rapidly adopted across various domains due to its decentralized architecture [43]. This innovative technology enables the creation of secure, distributed digital ledgers for recording transactions [11]. By utilizing advanced cryptographic methods, blockchain ensures the integrity and verification of each transaction [52, 64]. Within this ecosystem, smart contracts function as self-executing programs on the blockchain, automating the management of digital assets such as cryptocurrencies. These contracts are activated when specific conditions are met and, once deployed, become permanent components of the blockchain [73]. However, the immutability and inherent complexity of smart contracts pose significant security challenges [73]. The well-known DAO incident [7, 27] serves as a cautionary example, demonstrating the potential severity of such vulnerabilities. This security breach resulted in the illegal transfer of $60 million worth of Ethereum, causing widespread impact on the blockchain community [2].

Large Language Models (LLMs) have attracted widespread interest due to their strong natural language understanding and generation capabilities, showing impressive performance in various complex tasks, including code comprehension and generation [5, 22, 39, 44, 63, 67], and smart contract auditing and generation [29, 59–62]. The primary challenge is that existing Code LLMs often lack a deep, contextual understanding of security principles, leading to two interconnected failures. First, they fail to produce an explicit reasoning process, operating as "black boxes" that prevent developers from auditing the security logic. Second, this shallow understanding results in code with critical vulnerabilities. For instance, as illustrated in our motivating example (Fig. 1), a powerful model like `Qwen2.5-Coder-7B-Instruct`, when tasked to implement a `withdraw` function that prioritizes a developer fee, produces a flawed implementation. It correctly transfers the fee but then transfers the original, full balance to the recipient without deducting the fee, creating a severe "double payment" vulnerability. This error vividly demonstrates that even top-tier models cannot be trusted for secure contract generation without a more structured reasoning capability. This is not an isolated issue; empirical data from our broader experiments further substantiates that reasoning-focused models consistently outperform their standard counterparts. For instance, the reasoning-based `QwQ-32B` achieves a FullRate of 23.94%, surpassing the 20.50% achieved by the standard `Qwen2.5-32B-Instruct`. This pattern highlights that models equipped with explicit reasoning capabilities are better suited for the security-critical domain of smart contract generation.

Previous research has attempted to mitigate these risks from two main directions. The first is post-hoc vulnerability detection, which audits the code after it has been generated. Although recent work has leveraged Large Language Models (LLMs) for automated auditing with tools like GPTScan [42] and iAudit [26], this paradigm has a fundamental weakness. It decouples code generation from security verification, forcing developers to first receive potentially black-box-generated code and then use a separate tool to find its flaws. This not only disrupts the development workflow but can also lead to complex and inefficient fixes, especially when the auditing and generation models are inconsistent (a risk highlighted by iAudit's separated architecture [26]). This approach is inherently reactive, aiming to 'patch insecurity' rather than proactively 'build in security'. The second direction focuses on proactive secure code generation. For instance, Storhaug et al. [41] proposed a method based on vulnerability tagging and constrained decoding to proactively avoid

known insecure patterns during generation, while CodeBC [47] employs a three-stage fine-tuning strategy to enhance the security and practicality of the generated code. However, these approaches still often lack the explicit, multi-step reasoning required to handle complex business logic securely and fail to produce auditable thought processes. This leaves a critical gap for a model that not only generates secure code but also transparently explains how it achieved that security.

To bridge these gaps, we propose SmartCoder-R1 based on Qwen2.5-Coder-7B, a framework designed to generate smart contracts that are not only secure and functional but also inherently explainable by tackling the dual challenges of opaque reasoning and code vulnerability. Our approach is built on a meticulously designed three-stage training pipeline that ensures a logical closed loop from problem to solution. First, to build a foundational understanding of Solidity, Continual Pre-training (CPT) is performed on a comprehensive corpus of 286,397 instances. Second, to address the lack of transparency, we employ Long Chain-of-Thought Supervised Fine-Tuning (SFT) on 7,998 expert-validated samples. This stage explicitly teaches the model to emulate a security expert's thought process, generating a step-by-step reasoning chain (`<think>...</think>`) before writing the final code (`<answer>...</answer>`). Finally, to directly minimize vulnerabilities, we apply Security-Aware Group Relative Policy Optimization (S-GRPO) using 1,691 samples. This stage operates within a reinforcement learning paradigm, aligning the model (the policy network π_θ) with a programmatic reward function R. For each input, the policy network generates a group of candidate outputs. The reward function R, a weighted sum of binary scores for compilation success (R_{compile}), security compliance (R_{security}), and format correctness (R_{format}), then evaluates each output. The core of S-GRPO lies in calculating a normalized advantage for each candidate relative to the group's average performance and using this signal to update the model's parameters θ via a policy gradient update. By maximizing the expected reward, this method directly steers the model's generation distribution towards verifiably secure, compilable, and well-structured code, thereby internalizing complex security constraints into the model's generation policy.

To validate our approach, we conducted extensive experiments on a challenging benchmark comprising 756 real-world functions sourced from 289 deployed contracts, covering a wide range of complexities from simple utilities to intricate business logic. The results confirm the superiority of SmartCoder-R1. It establishes a new state of the art, achieving top performance across five key metrics: a ComPass (proportion of successfully compiled code) of 87.70%, a VulRate (proportion of compilable code with vulnerabilities; lower is better) of 8.60%, and a SafeAval (proportion of all generated code that is both compilable and secure) of 80.16%. Consequently, our model achieves a FuncRate (proportion of all generated code that is both compilable and functionally correct) of 53.84%, culminating in a final FullRate (the overall proportion of code that is simultaneously compilable, secure, and functionally correct) of 50.53%. This FullRate represents a substantial absolute improvement of 15.87 percentage points and a 45.79% relative improvement over the strongest reasoning-based baseline, DeepSeek-R1, which scored 34.66%. Furthermore, human evaluation of the reasoning process confirms its superiority in explainability. SmartCoder-R1's reasoning achieved high-quality ratings across all key dimensions of Functionality (82.7%), Security (85.3%), and Clarity (90.7%), comprehensively outperforming baselines.

The main contributions of this paper are as follows:

- We propose SmartCoder-R1, the first framework to systematically integrate Continual Pre-Training (CPT), Long Chain-of-Thought Supervised Fine-Tuning (L-CoT SFT), and Security-Aware Group Relative Policy Optimization (S-GRPO) for secure and explainable smart contract generation.

- We construct and release novel, high-quality datasets with expert-validated reasoning chains, comprising 7,998 samples for SFT and 1,691 samples for S-GRPO, to foster community research in explainable smart contract generation.
- We demonstrate that SmartCoder-R1 establishes a new state of the art, achieving a FullRate of 50.53% and outperforming the strongest baseline by a relative margin of 45.79%, proving its superior ability to generate verifiably secure and functional smart contract.

2 Background and Motivation

2.1 Problem Statement

Given a smart contract context C (defined as the minimal Solidity code environment required for compilation, including the contract's state variables, inherited contracts, function signatures excluding the target function F, modifiers, events, and library/struct definitions) and a functional requirement description R, the goal of smart contract code generation is to generate a function implementation F such that F satisfies the following constraints: **Functional Correctness**: F must correctly implement all functional requirements specified in R. This includes proper handling of input parameters, correct state variable modifications, appropriate return values, and adherence to the specified business logic. **Compilability**: F must be compilable by the Solidity compiler, i.e., $\text{Compilable}(C \cup F) = \text{True}$, adhering to language syntax, type rules, and correct integration with the contract context C. **Security**: F must not contain known security vulnerability patterns, i.e., $\text{Secure}(F) = \text{True}$. This encompasses critical security requirements such as reentrancy protection (e.g., proper state updates before external calls) and access control enforcement (e.g., appropriate permission checks). The complete taxonomy of twelve vulnerability classes that is systematically detailed in Section 3.1 and empirically evaluated in RQ5 (Table 6)

Necessity of Formal Constraints. These constraints are not merely desirable but hard requirements in smart contract development due to immutability and financial risks (e.g., the DAO attack's $60M loss [7, 27]). The formal specification serves three purposes: (1) *Problem specification*: distinguishing smart contract generation from general code synthesis where security is optional; (2) *Training justification*: our pipeline directly targets these constraints (CPT for syntactic patterns in $\text{Compilable}(C \cup F)$, L-COT SFT for reasoning across all three via expert-validated samples, S-GRPO for optimizing $\text{Compilable}(C \cup F)$ and $\text{Secure}(F)$ while maintaining Functional Correctness through KL-regularization); (3) *Evaluation grounding*: metrics map to constraints (ComPass measures Compilable, SafeAval measures Compilable $\wedge$ Secure, FullRate measures all three jointly).

2.2 Motivations

Summary. Smart contract development demands rigorous security due to the immutable and high-stakes nature of deployed assets [59–62, 64–66]. However, existing Code LLMs often fail to address this, treating contract generation as a standard coding task without the necessary structured reasoning. This oversight leads to two distinct limitations that motivate our work: (1) the absence of explicit, auditable security reasoning (**a process issue**), leaving developers with unverifiable "black box" code **that prevents them from understanding how security decisions were made**, and (2) as a consequence, the resulting generation of critical vulnerabilities stemming from a shallow understanding of contract-specific principles like fund conservation (**an outcome issue**).

Motivation 1: Lack of Explicit Security Reasoning. Most existing Code LLMs lack reasoning steps and simply treat smart contract generation as traditional code generation tasks. However, the immutable nature of smart contracts after deployment indicates that they require a higher level of security than traditional code generation tasks, as vulnerabilities can cause serious financial losses

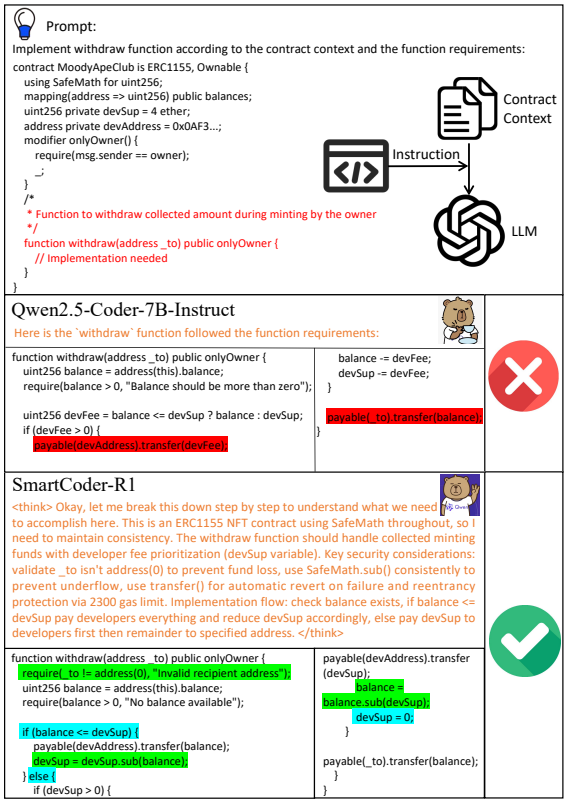


Fig. 1. A motivating example to illustrate the limitations of non-reasoning Code LLMs in secure smart contract generation, demonstrating the advantages of reasoning-enhanced Code LLMs.

[62]. Therefore, smart contract generation tasks need to consider not only functional requirements but also security requirements.

Security reasoning in smart contract development involves multi-layered processes: threat modeling (identifying attack vectors and malicious actors), state analysis (understanding security implications of state changes), interaction pattern analysis (considering secure interactions with other contracts and external calls), and economic incentive analysis (evaluating tokenomics security). Existing models like Qwen2.5-Coder and CodeLLaMA lack structured reasoning and generate code directly without demonstrating security decision-making. This "black box" approach prevents developers from verifying security assumptions. As Fig. 1 shows, Qwen2.5-Coder-7B-Instruct outputs code without reasoning, making security principles and risks opaque. This is particularly dangerous given smart contracts' immutability and potential for permanent financial losses. In contrast, SmartCoder-R1 provides explicit reasoning: "use SafeMath to prevent underflow, use transfer() to handle reentrancy attacks, implement developer fee priority logic, ensure fund conservation principles." This transparency enables developers to understand, verify, and audit code security rather than blindly trusting outputs.

Motivation 2: Critical Security Vulnerabilities in Generated Smart Contract. Taking the MoodyApeClub withdraw function in Fig. 1 as an example, code generated by Qwen2.5-Coder-7B-Instruct appears functionally correct but contains a critical business logic error: a fund double payment vulnerability. The code calculates developer fee `uint256 devFee = balance <= devSup ? balance : devSup`, pays it via `payable(devAddress).transfer(devFee)`, but then executes `payable(_to).transfer(balance)`, transferring the original complete balance to the recipient without deducting the paid devFee, causing total payments to exceed available funds.

This error manifests three security failures: fund integrity violation (transfers exceed available balance), business logic error (violates developer fee priority), and state consistency breakdown (contract state mismatches fund allocation). In MoodyApeClub NFT revenue distribution, developers receive fees while artists and promoters extract full shares, depleting contract funds. Such errors often pass basic syntax checks and simple functional tests because code structure appears reasonable, but cause serious financial losses in complex scenarios. Qwen2.5-Coder-7B-Instruct fails to understand contextual requirements and and priority relationships between different stakeholders in the smart contract ecosystem.

In contrast, SmartCoder-R1 identified this through explicit security reasoning, mentioning the need to "implement developer fee priority logic, ensure fund conservation principles." It generated the correct implementation: handling situations through `if (balance <= devSup)` branches, using `devSup = devSup.sub(balance)` to update remaining developer fees in the first branch, and `balance = balance.sub(devSup)` to correctly deduct paid fees from the balance in the second branch, then executing `payable(_to).transfer(balance)` to transfer the updated balance. This ensures fund conservation and avoids double payment.

This highlights current models' limited grasp of contract constraints (fund conservation, state consistency, multi-party balance). SmartCoder-R1 proves that explicit security reasoning is essential, as direct generation falls short in security-critical domains. Table 1 substantiates this: reasoning-enhanced models like DeepSeek-R1 (34.66% FullRate) and QwQ-32B (23.94%) significantly outperform their standard counterparts (DeepSeek-V3 at 30.16% and Qwen2.5-32B at 20.50%), confirming that reasoning capabilities yield more functional and secure code.

3 Approach

The *Approach* section details our end-to-end methodology for Dataset Construction, CPT, L-COT SFT, and Reinforcement Learning with S-GRPO. The Qwen2.5-Coder-7B serves as our base model, selected for its superior coding capabilities [14] and the Qwen family's superior adaptability to reinforcement learning (RL) compared to counterparts like Llama [50].

Pipeline Overview: Our three-stage training operates sequentially on Qwen2.5-Coder-7B (Fig. 2). (1) CPT: input 286,397 Solidity code instances → output domain-specialized base model. (2) L-CoT SFT: input 7,998 expert-validated samples (instruction + contract context + reasoning + code) → output reasoning-capable model. (3) S-GRPO: input 1,691 RL training samples → output final SmartCoder-R1 with security-aligned policy.

3.1 Dataset Construction

We construct the SFT and S-GRPO datasets through a multi-step process combining deterministic programmatic extraction, LLM-based data generation, and manual expert validation. Raw contract data is sourced from Ethereum mainnet, filtered to include only contracts with verified source code and function-level documentation. For each contract, we parse class-level and function-level code, extracting function signatures, documentation strings, and the entire Solidity context required for compilation. Where documentation is in Chinese, we translate it to English using DeepSeek-V3,

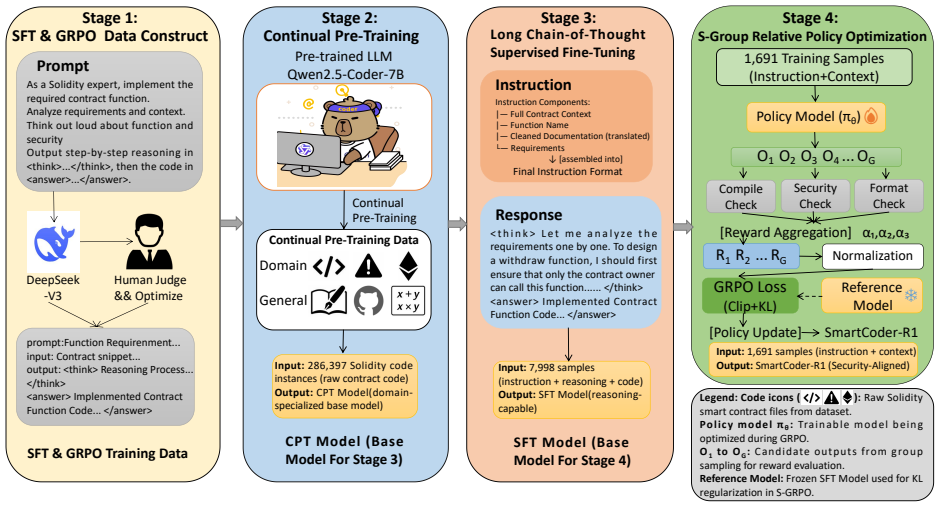


Fig. 2. Overview of our SmartCoder-R1 pipeline.

with fallback to a pattern-based description generator if translation fails. All documentation is cleaned to remove escape characters, annotation tags, and non-informative boilerplate.

For each function, we generate an instruction that specifies the target function name and the associated requirements. We then construct an LLM prompt that includes the full contract context, the function name, the cleaned and translated documentation, and a reference implementation if available. We use DeepSeek-V3 with a conversational, chain-of-thought prompt to generate a sample that includes a detailed, stepwise reasoning process in `<think>...</think>` tags and a standalone, syntactically correct Solidity implementation in `<answer>...</answer>` tags. The LLM is instructed to consider twelve specific vulnerability classes (detailed in Table 6, RQ5): reentrancy vulnerabilities, array bounds issues, access control flaws, state validation gaps, integer overflow/underflow (for Solidity <0.8.0), improper error handling, timestamp dependence, gas limit DoS risks, function visibility issues, tx.origin authentication, selfdestruct usage, and delegatecall context risks. While these twelve types cover the most critical and common vulnerabilities based on established security research, the model can potentially identify emerging patterns or context-specific risks beyond this taxonomy, such as flash loan attack vectors in DeFi protocols or governance mechanism exploits.

Each LLM-generated sample is post-processed to extract the reasoning chain (via regex for `<think>` tags), implementation code (from `<answer>` tags), and security-sensitive patterns (using fixed regular expressions for Solidity constructs and keywords). All samples are then reviewed by human experts experienced in Solidity security auditing. Reviewers verify that generated code compiles with the Solidity compiler (`solc`), addresses all stated requirements, and introduces no pre-defined security vulnerabilities. The review process includes manual reading and static analysis through professional-grade security scanning tools. Samples that fail to compile, fail security checks, or exhibit incomplete reasoning are corrected or removed. Importantly, while DeepSeek-V3 generates initial drafts, expert validation corrects all errors and refines security reasoning to ensure training quality exceeds the teacher's outputs; combined with novel security-oriented RL objectives (S-GRPO in Section 3.4), SmartCoder-R1 learns beyond its teacher, as demonstrated in Figure 4 where our model corrects DeepSeek-R1 (the reasoning-optimized version of DeepSeek-V3) errors.

3.2 Continual Pre-training

Stage Configuration: Base Model: Qwen2.5-Coder-7B; Input: 286,397 Solidity code instances (pure contract code); Output: CPT Model (domain-specialized base model for Stage 3).

During continual pre-training, we use only pure Solidity code, excluding comments and documentation, to prevent the model from memorizing non-functional text and to focus the learning on syntactic and structural patterns. Each contract is decomposed into a normalized format with explicit separation of contract-level declarations, state variables, modifiers, events, and function implementations. We segment each contract into overlapping code windows of up to 2048 tokens. For contracts exceeding the model's context window, we apply a sliding-window approach with 2,048 tokens per segment and overlap to preserve contextual continuity. While dependency-based segmentation could provide semantic boundaries, Solidity's dynamic dispatch mechanisms (e.g., `delegatecall`, interface calls) make static dependency graphs incomplete. The sliding-window approach naturally preserves local context (function signatures, state variables) and scales efficiently for large-scale processing. Each window is used as a training instance for next-token prediction.

The training objective is the standard left-to-right language modeling loss:

$$\mathcal{L}_{\text{CPT}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{T_i}\log P(x_{i,j} \mid x_{i,<j}) \tag{1}$$

where N is the total number of code windows in the pre-training corpus, T_i is the length of the i-th window, and $x_{i,j}$ is the j-th token in window i. We train the model for two epochs using the AdamW optimizer with a learning rate of 1×10^{-5}, batch size of 64, and a sequence length cutoff of 2048 tokens. Model checkpoints are saved every 500 steps. During pre-training, the model is exposed to contract patterns including inheritance, state variable initialization, error handling via `require` and `revert`, and common access control idioms. No reasoning or documentation information is present in this phase.

3.3 Long Chain-of-Thought Supervised Fine-Tuning

Stage Configuration: Base Model: CPT Model; Input: 7,998 expert-validated samples (instruction + contract context, `<think>` reasoning + `<answer>` code); Output: SFT Model.

The purpose of the Long Chain-of-Thought (CoT) Supervised Fine-Tuning stage is to explicitly endow the model with the capacity to perform multi-step reasoning required for secure smart contract code generation. Each training sample consists of an instruction, contract context, a detailed stepwise reasoning chain enclosed in `<think>...</think>`, and a standalone Solidity implementation enclosed in `<answer>...</answer>`. The output sequence is strictly required to begin with a multi-sentence, logically ordered reasoning trace, followed by a complete function implementation. The supervised fine-tuning loss is:

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{M}\sum_{k=1}^{M}\sum_{l=1}^{L_k}\log P(y_{k,l} \mid I_k, X_k, y_{k,<l}) \tag{2}$$

where M is the number of samples, I_k is the instruction, X_k is the input, $y_{k,l}$ is the l-th output token. The model is trained for three epochs, learning rate 1×10^{-5}, batch size 8, and maximum sequence length 8192. All outputs are required to compile with `solc` and to address at least one specified vulnerability class. This stage imparts the model with the initial ability to decompose business requirements into explicit security logic steps and to realize these as functional, secure, and compilable code. After SFT, the model can generate code that is not only correct and complete, but also accompanied by an auditable reasoning trace, providing the foundation for downstream RL alignment.

3.4 Security-Aware Group Relative Policy Optimization

Stage Configuration: Base Model: SFT Model; Input: 1,691 RL training samples (instruction + contract context); Output: SmartCoder-R1 (security-aligned final model).

The S-GRPO stage acts as a targeted refinement process, aligning the model with the complex, multi-faceted objectives of real-world smart contract development that are difficult to capture with supervised learning alone. For each input consisting of a contract's existing code (`context`) and a new functional requirement, the model generates a response containing: (1) a **Long Chain-of-Thought** (`<think>`) reasoning process that explicitly outlines security considerations and implementation plan, and (2) the final, compilable **Solidity function** (`<answer>`).

During training, we sample a group of G candidates from the current policy network for each prompt. Each candidate is then subjected to a rigorous, automated evaluation to calculate its reward:

(1) **Compilation Check (R_{compile}):** This is the most foundational prerequisite for any smart contract. The generated Solidity code is compiled using `solc`. A successful compilation ($R_{\text{compile}} = 1$) signifies that the code is syntactically valid and can be deployed on the blockchain. A failure ($R_{\text{compile}} = 0$) indicates a fundamental flaw, rendering the output useless.

(2) **Security Rule Check (R_{security}):** This is the most critical component, directly targeting the core challenge of secure contract generation. Our static analysis targets the 12 vulnerability types in Table 6 (RQ5) via a dual-layer approach: **(1) Regex-based** rules detect syntactic anti-patterns at all generation stages, including partially-generated code; The complete set of 12 regex patterns is available in our replication package (Section 8). **(2) Slither AST-based** detectors perform semantic analysis on compiled code, mapping to vulnerability types as follows: reentrancy (reentrancy-eth, reentrancy-no-eth, reentrancy-benign, reentrancy-events), array bounds (array-by-reference, controlled-array-length), access control (arbitrary-send-eth, unprotected-upgrade), state validation (unchecked-transfer, divide-before-multiply, missing-zero-check), integer overflow (Solidity version `<0.8.0` detection), error handling (unchecked-lowlevel, unchecked-send), timestamp dependence (timestamp, weak-prng), gas limit DoS (costly-loop, reentrancy-unlimited-gas), visibility issues (external-function), tx.origin (tx-origin), selfdestruct (suicidal), and delegatecall risks (controlled-delegatecall, delegatecall-loop). Both outputs merge via OR-gate logic in R_{security} computation: code is flagged as vulnerable if either layer detects an issue. These checks operate conservatively ('better safe than sorry'), prioritizing high recall to minimize false negatives, accepting occasional false positives (over-flagging) rather than missing vulnerabilities. While no static analysis can completely avoid false negatives, our dual-layer approach achieved 0% false negatives across 100 expert-validated samples (Section 4.4) for the 12 targeted vulnerability classes (RQ5), demonstrating that this conservative strategy effectively reduces missed vulnerabilities. For example, when generating a `withdraw` function, the **regex layer** first scans for `.call{value:` patterns and verifies **Checks-Effects-Interactions (CEI)** compliance syntactically, while **Slither's analysis** confirms no state variables are modified after external calls. Similar dual-layer checks apply to all 12 vulnerability types listed above, including access control enforcement, `delegatecall` safety, and integer overflow prevention. A secure output receives $R_{\text{security}} = 1$; otherwise, it receives $R_{\text{security}} = 0$.

(3) **Format Compliance (R_{format}):** This reward ensures the model maintains its explainability, a cornerstone of our approach. The output must contain a well-structured `<think>` block with substantive reasoning (at least three steps) followed by a distinct `<answer>` block. This structure provides a crucial **cognitive scaffold** for developers, allowing them to audit the model's security logic before trusting its code. A compliant format yields $R_{\text{format}} = 1$.

The total reward R for each candidate is a weighted sum of these three binary scores:

$$R = \alpha_1 R_{\text{compile}} + \alpha_2 R_{\text{security}} + \alpha_3 R_{\text{format}} \tag{3}$$

We set the weights to $(\alpha_1, \alpha_2, \alpha_3) = (0.3, 0.5, 0.2)$, prioritizing security (R_{security}) above all else, as a single vulnerability can compromise an entire contract.

The policy is updated using the S-GRPO objective, which combines the clipped policy ratio surrogate loss with a KL penalty. The overall S-GRPO loss is:

$$J_{\text{S-GRPO}}(\theta) = \mathbb{E}_{q,\{o_i\}_{i=1}^{G}} \left[\frac{1}{G} \sum_{i=1}^{G} \frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \min \left(\frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})} \hat{A}_{i,t}, \right. \right.$$
$$\left. \left. \text{clip} \left(\frac{\pi_\theta(o_{i,t}|q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q, o_{i,<t})}, 1-\epsilon, 1+\epsilon \right) \hat{A}_{i,t} \right) \right] - \beta D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}] \tag{4}$$

where $\hat{A}_{i,t}$ is the normalized advantage, and β is the KL penalty coefficient. The KL divergence $D_{\text{KL}}[\pi_\theta \| \pi_{\text{ref}}]$ is estimated for every token in the output sequence using an unbiased estimator.

Concretely, during each RL step, the model receives as input the full contract context and functional requirement, and must generate a reasoning chain that explicitly discusses security best practices and then output a function that is both functionally correct and secure. The reward system ensures that the model only receives a high score if the code passes full compilation, contains no statically detectable vulnerabilities, and the reasoning is explicit and well-structured. RL training is performed for 5 epochs, with all outputs and reward components logged.

In essence, S-GRPO constructs an efficient, automated "trial-and-error" learning environment. The core idea is teaching the model to write secure smart contracts through group-based comparison and immediate feedback. In practice, when faced with a programming task (e.g., implementing a re-entrancy-proof `withdraw` function), the model generates a diverse group of potential solutions. Some may be perfect, others might contain security vulnerabilities, and some may not compile. The system acts as an automated "code reviewer," rigorously evaluating each attempt: solutions that compile successfully, adhere to secure design patterns (like Checks-Effects-Interactions), and provide clear reasoning receive high rewards (high advantage). Conversely, those with vulnerabilities or errors receive low rewards or penalties (low advantage). This reward signal directly guides parameter updates, reinforcing behaviors that lead to high-quality outputs while discouraging problematic ones. Through continuous, automated iteration, S-GRPO compels the model to learn not only "how to write code," but more importantly, to internalize "why it must be written this way to be secure." The S-GRPO reward mechanism is modular. While targeting 12 general vulnerability classes (Table 6), practitioners can extend R_{security} with domain-specific rules (e.g., AMM invariant checks, oracle guards) via custom regex or Slither detectors to address business-logic vulnerabilities without retraining.

While R_{compile} ensures syntactic validity, functional correctness is guaranteed through three mechanisms: (1) L-CoT SFT's training on expert-validated, functionally correct implementations; (2) R_{security} and R_{format} implicitly validating functional completeness; (3) KL regularization anchoring the policy to the functionally correct SFT reference model. Execution-based testing (Section 4.6) rigorously validates this via unit tests.

4 Experiments

4.1 Research Questions

To evaluate our SmartCoder-R1, we conduct experiments to answer the following research questions:

- **RQ1**: How does our SmartCoder-R1 perform in generating secure and functional smart contracts compared to state-of-the-art code generation models?
- **RQ2**: How do individual components contribute to the overall performance, and how sensitive is performance to key parameter tuning?

• **RQ3**: How effective are the reasoning chains generated by SmartCoder-R1 in terms of their Functionality, Security, and Clarity?

• **RQ4**: How does SmartCoder-R1's reasoning and security awareness compare to state-of-the-art methods through representative case studies?

• **RQ5**: What are the primary vulnerability types in generated insecure contracts, and what factors contribute to these security failures?

4.2 Dataset

Continual Pre-training (CPT): Following Yu et al.[61], we construct our CPT dataset from 186,397 unique Ethereum smart contracts provided by Storhaug et al.[41] (501.62M tokens). Both sources are peer-reviewed works published at top-tier venues (ISSTA, ISSRE), ensuring data quality. To enhance diversity, we augment this with 100,000 instances of general code, mathematics, and texts in English and Chinese (118.94M tokens), creating a corpus of 286,397 instances totaling 620.56M tokens.

Long Chain-of-Thought SFT (L-COT SFT): Our SFT dataset features 7,998 meticulously annotated samples (16.44M tokens, avg. 2,056 per sample) from 1,508 unique contracts from Storhaug et al. [41]. Each contract contains multiple functions, which are expanded into function-level samples. It covers diverse patterns, including ERC20/721, Ownable, crowdsales, pausable/proxy patterns, AccessControl, multisig wallets, Governor/Timelock modules, and DeFi primitives. Crucially, each instance contains the full contract context, functional requirements, the target implementation, and an explicit Chain-of-Thought reasoning trace constructed by us. The function distribution is realistic (constructors 4.5%, regular 55.3%, view 35.2%, payable 5.1%) following the empirical distribution of deployed mainnet contracts [41]. Correctness was ensured through both manual verification and automated analysis.

Security-Aware Group Relative Policy Optimization (S-GRPO): The S-GRPO dataset is built from the same sources as the SFT stage but is strictly separated. It contains 1,691 samples (2.74M tokens, avg. 1,621 per sample), expanded at the function level from the source contracts, with 1,200 samples for training and 491 for evaluation. The different sample counts between SFT (7,998) and S-GRPO (1,691) arise because SFT focuses on comprehensive reasoning coverage across all major functions, while S-GRPO targets security-critical functions for reinforcement learning. Notably, we constructed new reasoning chains tailored for this RL stage. Each sample provides a contract context, requirement, reasoning chain, and labeled implementation to facilitate policy optimization with step-by-step feedback.

Evaluation: Our evaluation set comprises 756 unique functions from 289 real-world contracts, sourced from Storhaug et al. [41] and EtherScan, spanning 53 Solidity versions. It includes 335 distinct implementations, ranging from 44-character utilities to 3,149-character business logic. The dataset covers multi-layered structures, from libraries (SafeMath, EnumerableSet, SafeCast) to critical patterns (Ownable, Proxy, Roles), and features 88.10% high-quality NatSpec documentation.

Annotation Effort: Total manual validation effort was approximately 322-485 hours (SFT: 7,998 samples × 2-3 min/sample = 266-400 hours; S-GRPO: 1,691 samples × 2-3 min/sample = 56-85 hours). With six security experts, the per-person workload was manageable (54-81 hours each), ensuring thorough quality control while maintaining scalability.

Data Leakage Prevention: We enforced strict non-overlapping splits across CPT, SFT, S-GRPO, and Evaluation datasets, applying token-based Jaccard similarity deduplication (threshold 0.9) at the function level to remove near-duplicates, with the evaluation set fully isolated from all training and fine-tuning data to guarantee zero overlap. Although multiple datasets cite Storhaug et al. [41], they differ fundamentally in composition: CPT uses contract-level raw code; SFT and S-GRPO use expert-annotated instruction-response pairs with reasoning chains; and the evaluation set draws

function-level implementations from entirely different contracts, ensuring zero overlap despite the shared source.

4.3 Baselines

We evaluate our model against state-of-the-art baselines selected using three explicit criteria: (1) **Category Representativeness**: we include models from three categories to cover the full spectrum of current approaches; (2) **Reproducibility**: all baselines have either open-source weights, accessible APIs, or published reproducible implementations; (3) **State-of-the-Art Performance**: models were selected based on peer-reviewed publications, strong benchmark performance, and recent release dates (2023-2025). Based on these criteria, we selected baselines from three categories: **General/Code LLMs**: These are LLMs with strong, but not specialized, code generation capabilities. This category includes: the CodeLlama-Instruct series (7B, 13B, 34B) [38], the DeepSeek-Coder-Instruct series (6.7B, 33B) [10], the Qwen-Coder series (Qwen2.5-3B, 14B, 32B-Instruct [14]; Qwen3-30B-A3B-Instruct [56]), Qwen2.5-32B-Instruct [57], DeepSeek-V3 (DeepSeek-V3-0324) [18], the Llama-3 series (3.1-8B, 3.2-1B), and the commercial models GPT-4.1 (gpt-4.1-2025-04-14) [30]. **Domain-Specific Smart Contract LLMs**: These are LLMs specifically designed for smart contract generation. We include Storhaug et al. [41] and CodeBC [47], both state-of-the-art frameworks for this task. Since CodeBC is not fully open-source, we re-implemented its methodology based on the original paper and trained it using our own SFT dataset to ensure a fair comparison. **Reasoning LLMs**: These LLMs incorporate explicit multi-step reasoning. We compare against DeepSeek-R1 [9] and QwQ-32B [35], which serve as strong baselines for reasoning-enhanced generation.

4.4 Metrics

Following established evaluation frameworks [47], we use five primary metrics. We measure **ComPass(%)**, the proportion of successfully compiled code. As a conditional metric, **VulRate(%)** is the percentage of compilable code containing known vulnerabilities (lower is better). We then use three "yield" metrics: **SafeAval(%)** for code that is both compilable and secure, **FuncRate(%)** for code that is compilable and functionally correct, and **FullRate(%)**, the proportion meeting all criteria. Note that SafeAval and VulRate use different denominators and are not directly additive: ComPass and SafeAval are computed over all samples, while VulRate is computed only over compiled outputs. The correct relationship is SafeAval = ComPass × (1 - VulRate), not SafeAval + VulRate = ComPass. To rigorously evaluate **VulRate** and **SafeAval**, we combined automated scanning with expert auditing. First, all successfully compiled code was scanned with the **Slither** tool, which categorizes potential vulnerabilities by severity (e.g., High, Medium, Low). Due to the high cost of expert audits, we selected a sample of **100** code snippets for manual review. This selection prioritized snippets with High-severity warnings, followed by those with Medium-severity. The final **VulRate** is a calibrated metric. For SmartCoder-R1, our dual-layer system initially flagged 62 of 663 compilable samples as potentially vulnerable. We audited all 62 flagged samples plus 38 randomly selected unflagged samples. Expert review revealed 57 true positives and 5 false positives among flagged samples (8% false-positive rate), and 0 missed vulnerabilities among unflagged samples (0% false-negative rate for our 12 targeted vulnerability classes, Table 6). The final **VulRate** of 8.6% (57/663) is a calibrated metric representing only confirmed true positives. For **FuncRate**, we manually constructed a unit test suite based on the ground-truth implementations of the target functions. We then evaluated the functional correctness of the generated code by executing these test cases. **FuncRate** represents the proportion of generated functions that successfully pass all corresponding tests, ensuring they strictly meet the specified functional requirements. We avoid traditional similarity (e.g., BLEU [32], CodeBLEU [37]) and sampling metrics like pass@k; they merely capture surface resemblance or tolerate insecure code, which is insufficient for smart contract security.

To evaluate the quality of the generated reasoning chains (RQ3), we manually assessed them on a 4-point Likert scale [17] across three dimensions: Functionality (whether the reasoning correctly understands and plans for all required features), Security (whether all relevant security risks are identified), and Clarity (logical flow and readability of the explanation). The assessment was conducted by six senior experts, each with over five years of experience in Solidity development and security auditing, recruited from diverse institutions (two blockchain companies, one university research group, and two independent consultants) with no prior project relationship, ensuring balanced, unbiased perspectives. All evaluations employed strict double-blind protocols: (1) all 150 samples were randomized and anonymized with random identifiers (e.g., "Sample_A37") before distribution, with no indication of the source model (DeepSeek-R1, QwQ-32B, or SmartCoder-R1); (2) each sample was independently rated by two blinded experts with no knowledge of their peers' ratings; (3) model identities were disclosed only after all ratings were finalized. To minimize subjectivity, all six experts first calibrated by independently rating a pilot set of 30 samples, yielding a Fleiss' Kappa of 0.85 across all three dimensions, indicating substantial agreement. For the formal 150-sample evaluation, Fleiss' Kappa remained consistently high across all dimensions (Functionality: 0.87, Security: 0.89, Clarity: 0.88, overall average: 0.88), confirming reliable inter-rater consistency at scale. In cases where two experts' scores diverged by 2 or more points, a third expert blinded to the model source arbitrated to reach a consensus-based final rating. These comprehensive safeguards ensure that observed performance differences reflect genuine reasoning quality rather than evaluator bias or expectations.

4.5 Implementation Details

We perform CPT and L-COT SFT using LlamaFactory [71] and DeepSpeed [36] with `fp16` enabled. Loss is computed via cross-entropy, and parameters are optimized with AdamW [21] ($\beta = (0.9, 0.99)$, $\epsilon = 1e{-}8$). All models use full parameter tuning. **CPT**: batch size 64, gradient accumulation steps 16, epochs 2, learning rate 1e−5 (cosine decay), warmup steps 0, cutoff length 2048, save steps 500. **L-COT SFT**: batch size 8, gradient accumulation steps 8, epochs 3, learning rate 1e−5 (cosine decay), warmup steps 0, cutoff length 8192, save steps 50. **S-GRPO**: built upon Logic-RL [55] and VeRL [40], initialized from L-COT SFT checkpoint. Batch size 8, max prompt length 24,576, max response length 2,048, learning rate 3×10^{-7}. KL loss regularization (coef = 0.001). Mini-batch and micro-batch sizes 8. Gradient checkpointing and full offloading enabled. Parallel rollouts 8 (`rollout.n=8`), rollout/reference log probability micro-batch size 160. Training: single node, 8 NVIDIA H800 GPUs (80GB each), 5 epochs, checkpoints every 20 steps. **Evaluation**: greedy decoding (`do_sample=false`) for reproducibility and fair comparison across models. All experiments use identical hardware.

4.6 Experimental Results

In this section we present experimental results and analysis to answer the research question.

1) RQ1: As shown in Table 1, SmartCoder-R1 achieves the highest overall performance across all key metrics when evaluated against a comprehensive range of baselines.

Specifically, SmartCoder-R1 achieves a ComPass (compilability) of 87.70%, improving over the best general/code LLM (GPT-4.1 at 84.13%) by 4.24% relatively and over the best domain-specific model (Storhaug et al. at 82.67%) by 6.08% relatively. On the critical VulRate metric (lower is better), SmartCoder-R1 attains a remarkably low vulnerability rate of 8.60%, which is 49.05% lower than CodeBC (16.88%), 44.80% lower than Qwen2.5-Coder-32B-Instruct (15.58%), and 72.09% lower than GPT-4.1 (30.82%). For the SafeAval (compilable and secure) metric, SmartCoder-R1 achieves 80.16%, representing a 17.20% relative improvement over CodeBC (68.39%) and a 26.52% relative improvement over Storhaug et al. (63.36%). Furthermore, SmartCoder-R1 demonstrates notable

Table 1. RQ1: Comparison of Secure and Functional Smart Contract Generation. Performance (%) of SmartCoder-R1 and state-of-the-art code generation baselines on secure smart contract generation benchmarks. ↑ indicates higher is better, ↓ indicates lower is better. **Bold** highlights the best results in each column. *With explicit security prompts for all 12 vulnerability classes.

Model	ComPass ↑	VulRate ↓	SafeAval ↑	FuncRate ↑	FullRate ↑
General / Code LLMs					
CodeLlama-7b-Instruct	29.63	14.29	25.40	13.62	12.57
CodeLlama-13b-Instruct	54.63	23.24	41.93	25.13	21.83
CodeLlama-34b-Instruct	48.94	25.14	36.64	23.15	20.63
Llama-3.2-1B-Instruct	7.01	16.98	5.82	3.17	1.98
Llama-3.1-8B-Instruct	22.09	11.98	19.44	6.35	5.29
DeepSeek-Coder-6.7B-Instruct	15.08	21.05	11.90	4.89	4.37
DeepSeek-Coder-33B-Instruct	18.52	25.71	13.76	12.30	10.58
Qwen2.5-Coder-3B-Instruct	12.57	22.11	9.79	7.01	5.29
Qwen2.5-Coder-14B-Instruct	51.19	10.85	45.63	24.34	20.90
Qwen2.5-Coder-32B-Instruct	62.83	15.58	53.04	38.23	29.50
Qwen2.5-32B-Instruct	48.02	22.31	37.30	27.12	20.50
Qwen3-Coder-30B-A3B-Instruct	78.70	37.14	49.47	41.40	19.97
DeepSeek-V3	69.05	33.91	45.63	48.28	30.16
GPT-4.1	84.13	30.82	58.20	52.91	30.29
Domain-Specific Smart Contract LLMs					
Storhaug et al. [41]	82.67	23.36	63.36	34.79	23.54
CodeBC [47]	82.28	16.88	68.39	33.20	25.79
Reasoning LLMs					
DeepSeek-R1	84.39	32.29	57.14	50.66	34.66
QwQ-32B	67.46	44.12	37.70	40.48	23.94
SmartCoder-R1 (Ours)	**87.70**	**8.60**	**80.16**	**53.84**	**50.53**

improvements in FuncRate (functional correctness, 53.84%), slightly outperforming the strongest baselines (e.g., +1.76% relative over GPT-4.1 at 52.91%). Especially, SmartCoder-R1 reaches a FullRate (functional correctness, security, and compilability simultaneously) of 50.53%, reflecting a 45.79% relative improvement over the next best result (DeepSeek-R1 at 34.66%). The seemingly low VulRate of some smaller models is an artifact of their frequent failure to produce compilable code (e.g., incomplete snippets or plain text), which are excluded from analysis. Our ablation study shows removing reasoning components increases VulRate to 18.68% (Table 3), confirming that prompt engineering cannot substitute systematic reasoning-enhanced post-training and security-aware group relative policy optimization.

Statistical Validation. To rigorously assess the significance of our improvements, we conducted Mann-Whitney U tests comparing SmartCoder-R1 against two state-of-the-art baselines (DeepSeek-R1 and GPT-4.1) on two critical metrics, with Bonferroni correction for multiple comparisons (4 tests total, adjusted $\alpha = 0.01/4 = 0.0025$). Table 2 presents detailed statistical comparisons. All tests demonstrate strong significance ($p < 0.001$, well below the corrected threshold). Effect sizes (Cliff's Delta $\delta = 0.16$-0.24) are classified as *small* by traditional standards but inherently compressed for binary outcomes. The consistent substantial relative improvements across metrics provide strong evidence of genuine, robust advances rather than random variation.

Table 2. Statistical Comparison of SmartCoder-R1 vs. State-of-the-Art Baselines

Metric	Baseline	Value	Ours	Absolute Δ	Relative Δ	U-stat	Cliff's δ
FullRate	DeepSeek-R1	34.66%	50.53%	+15.87ppt	+45.79%	331,128	0.16
	GPT-4.1	30.29%		+20.24ppt	+66.82%	343,602	0.20
VulRate	DeepSeek-R1	32.29%	8.60%	−23.69ppt	−73.37%	261,603	0.24
	GPT-4.1	30.82%		−22.22ppt	−72.10%	257,682	0.22

Note: All p-values < 0.001 (Bonferroni-corrected $\alpha = 0.0025$). Effect sizes are "small" by traditional standards but correspond to substantial practical improvements in binary success/failure outcomes.

Answer to RQ1: SmartCoder-R1 achieves state-of-the-art performance in generating secure smart contracts that meet functional requirements, outperforming the strongest baseline across all key metrics (ComPass, VulRate, SafeAval, FuncRate, FullRate). Statistical tests confirm highly significant improvements ($p < 0.001$) with substantial relative gains on production-critical metrics (FullRate and VulRate).

2) RQ2: Our ablation study in Table 3 systematically dissects the contribution of each core component: CPT, L-COT SFT, and S-GRPO to the overall performance.

Table 3. Ablation Study: Impact of Different Training Components. Performance (%) of SmartCoder-R1 variants with different training configurations. w/o: without.

Model Variant	ComPass ↑	VulRate ↓	SafeAval ↑	FuncRate ↑	FullRate ↑
w/o CPT & S-GRPO	80.82	19.80	64.81	31.22	20.37
w/o CPT & L-COT SFT	24.60	9.68	22.22	15.08	14.55
w/o CPT	85.05	14.15	73.02	37.17	31.75
w/o S-GRPO	84.26	18.68	68.52	32.67	24.21
w/o L-COT SFT	67.86	27.10	49.47	46.96	32.67
Full Model	**87.70**	**8.60**	**80.16**	**53.84**	**50.53**

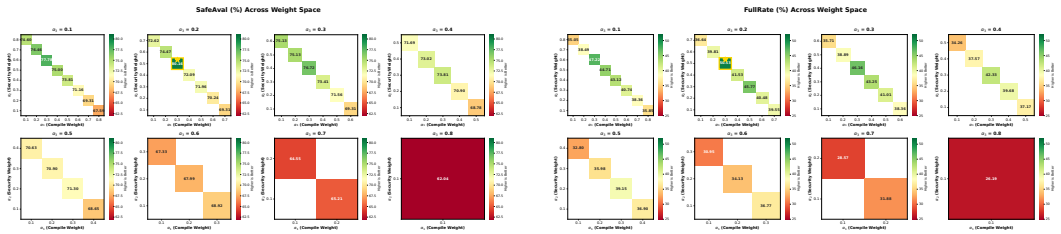
Table 4. Parameter Sensitivity Analysis Results.

Configuration	Weights $(\alpha_1, \alpha_2, \alpha_3)$	ComPass (%) ↑	VulRate (%) ↓	SafeAval (%) ↑	FuncRate (%) ↑	FullRate (%) ↑
Security+	(0.2, 0.6, 0.2)	86.51	13.91	74.47	47.09	39.81
Security++	(0.1, 0.7, 0.2)	85.71	15.28	72.62	45.90	36.64
Compile+	(0.4, 0.4, 0.2)	85.85	16.02	72.09	49.21	41.53
Compile++	(0.5, 0.3, 0.2)	85.32	15.66	71.96	54.10	45.77
Compile+++	(0.6, 0.2, 0.2)	84.39	16.77	70.24	50.40	40.48
Compile++++	(0.7, 0.1, 0.2)	85.71	19.14	69.31	49.21	39.55
Ours	(0.3, 0.5, 0.2)	**87.70**	**8.60**	**80.16**	53.84	**50.53**

The baseline (`w/o CPT & S-GRPO`) achieves `ComPass` of 80.82% but suffers from high `VulRate` (19.80%) and low `FullRate` (20.37%), showing that reasoning-and-code format alone cannot ensure

security. Adding CPT (`w/o S-GRPO`) improves metrics (`ComPass` 84.26%, `SafeAval` 68.52%) through better Solidity understanding, yet `VulRate` remains high (18.68%), confirming this combination cannot fully solve the security challenge. The critical breakthrough comes from S-GRPO: comparing the full model to `w/o S-GRPO`, `VulRate` drops from 18.68% to 8.60% (53.96% relative reduction), `SafeAval` rises from 68.52% to 80.16%, and `FullRate` jumps from 24.21% to 50.53%, demonstrating S-GRPO's power to directly optimize complex objectives like security and compilability.

The `w/o L-COT SFT` variant (CPT + S-GRPO) achieves surprisingly high `FuncRate` (46.96%), nearly matching the full model, suggesting S-GRPO's powerful optimization can directly steer the model towards functional correctness with CPT's strong foundation. However, this comes at a steep cost: its `VulRate` is highest among primary variants at 27.10%, starkly illustrating L-COT SFT's indispensable role. L-COT SFT teaches crucial structured reasoning (`<think>...</think>`) that acts as a cognitive scaffold, guiding the model to explicitly consider security constraints *before* code generation, making subsequent S-GRPO refinement far more effective and targeted. Without this scaffold, the model achieves functional correctness but frequently fails to navigate complex security requirements, producing insecure implementations.



(a) SafeAval (%) across weight space (b) FullRate (%) across weight space

Fig. 3. **Comprehensive Grid Search Heatmaps.** Systematic evaluation of all 36 configurations ($\alpha_1+\alpha_2+\alpha_3 = 1$, 0.1 increments). Each subplot shows a fixed α_3 (format weight), with α_1 (compile) on x-axis and α_2 (security) on y-axis. Star marks the global optimum (0.3, 0.5, 0.2). Darker colors indicate worse performance.

To further investigate the impact of S-GRPO, we conducted a parameter sensitivity analysis on the reward weights, with results presented in Table 4. This analysis reveals the delicate balance required between compilation (α_1), security (α_2), and format (α_3). Configurations that over-prioritize security (e.g., 'Security++' with $\alpha_2 = 0.7$) paradoxically result in a higher vulnerability rate (15.28%) compared to our balanced approach (8.60%). Conversely, increasing the weight on compilation (e.g., 'Compile++' with $\alpha_1 = 0.5$) boosts functional correctness to 54.10% but at the expense of security (VulRate of 15.66%). Our chosen configuration ($\alpha_1 = 0.3, \alpha_2 = 0.5, \alpha_3 = 0.2$) strikes an optimal trade-off by effectively harmonizing all three objectives, achieving the highest FullRate (50.53%). To validate our reward system's effectiveness, we analyze Table 4's systematic patterns. Increasing α_1 from 0.4 to 0.7 raises VulRate from 16.02% to 19.14% (+19.4%) while maintaining stable ComPass (~85%), confirming R_{compile} shapes the compilation-security trade-off. Meanwhile, α_2 exhibits a U-shaped effect: moderate values (0.5–0.6) achieve optimal safety (VulRate 8.60%–13.91%), while extremes degrade performance via under-prioritization ($\alpha_2 \leq 0.2$, VulRate ~17–19%) or over-fitting ($\alpha_2 = 0.7$, VulRate 15.28%). Our configuration (0.3, 0.5, 0.2) resides at the intersection of these patterns, achieving the best overall performance (VulRate 8.60%, SafeAval 80.16%).

Comprehensive Grid Search Validation. To rigorously validate that our optimal configuration is not cherry-picked but systematically identified, we conducted an exhaustive grid search covering all 36 valid weight combinations with 0.1 increments (satisfying $\alpha_1+\alpha_2+\alpha_3 = 1$). Fig. 3 visualizes the complete performance landscape across the weight space. The heatmaps reveal clear performance

gradients, with our configuration (0.3, 0.5, 0.2, marked with a star) residing at the global optimum. Notably, SafeAval (Fig. 3a) peaks sharply at this point (80.16%), while FullRate (Fig. 3b) shows similar dominance (50.53% vs. 47.22%). The systematic exploration confirms three key findings: (1) a precise 3:5 ratio between α_1 and α_2 yields synergistic effects, where security and functionality simultaneously peak rather than trade-off; (2) format weight α_3 exhibits an inverted-U relationship, with $\alpha_3 = 0.2$ providing optimal cognitive scaffolding without over-constraining the model; and (3) the optimal region demonstrates cross-slice consistency, with $\alpha_1 = 0.3$ maintaining near-optimal performance across multiple α_3 values, confirming the robustness of our findings.

Answer to RQ2: The ablation study demonstrates that CPT, L-COT SFT, and S-GRPO are all indispensable. CPT provides foundational domain knowledge, L-COT SFT establishes the crucial reasoning-to-code structure, and S-GRPO acts as a refiner that optimizes for security and correctness, leading to a synergistic effect that is essential for overall performance. The parameter sensitivity analysis further confirms that the effectiveness of S-GRPO is highly dependent on a balanced reward configuration.

3) RQ3: We had human experts rate reasoning chains of SmartCoder-R1 and two baselines on 150 samples using a 4-point Likert scale for Functionality, Security, and Clarity. As shown in Table 5, SmartCoder-R1 demonstrates superior performance across all dimensions. For **Functionality**, SmartCoder-R1 achieves high-quality ratings (scores 3 or 4) in 82.67% of cases (124/150), significantly outperforming DeepSeek-R1 (76.00%) and QwQ-32B (59.33%), indicating enhanced ability to interpret complex requirements and formulate correct implementation plans. The most significant advantage appears in **Security**, where our model receives high-quality ratings in 85.33% of samples (128/150), far exceeding DeepSeek-R1 (54.67%) and QwQ-32B (40.67%). This highlights our training paradigm's effectiveness in instilling deep security understanding, critical for smart contract generation. For **Clarity**, SmartCoder-R1 leads with 90.67% high-quality ratings, making reasoning chains easier to audit and trust compared to DeepSeek-R1 (85.33%) and QwQ-32B (69.33%).

Table 5. Human Evaluation of Reasoning Chain Quality (RQ3). Based on 150 samples, human experts rated the reasoning chains from different models. The scores are on a 4-point Likert scale (1=Poor, 2=Fair, 3=Good, 4=Excellent). The values in the table represent the number of samples receiving each score.

Model	Dimension	1 (Poor)	2 (Fair)	3 (Good)	4 (Exc.)
DeepSeek-R1	Functionality	9	27	73	41
	Security	21	47	56	26
	Clarity	6	16	79	49
QwQ-32B	Functionality	17	44	66	23
	Security	36	53	47	14
	Clarity	11	35	73	31
SmartCoder-R1 (Ours)	Functionality	**7**	**19**	**56**	**68**
	Security	**6**	**16**	**56**	**72**
	Clarity	**3**	**11**	**57**	**79**

Answer to RQ3: The reasoning chains generated by SmartCoder-R1 demonstrate superior quality in functionality, security, and clarity compared to strong reasoning-focused baselines, making them more reliable for secure smart contract generation.

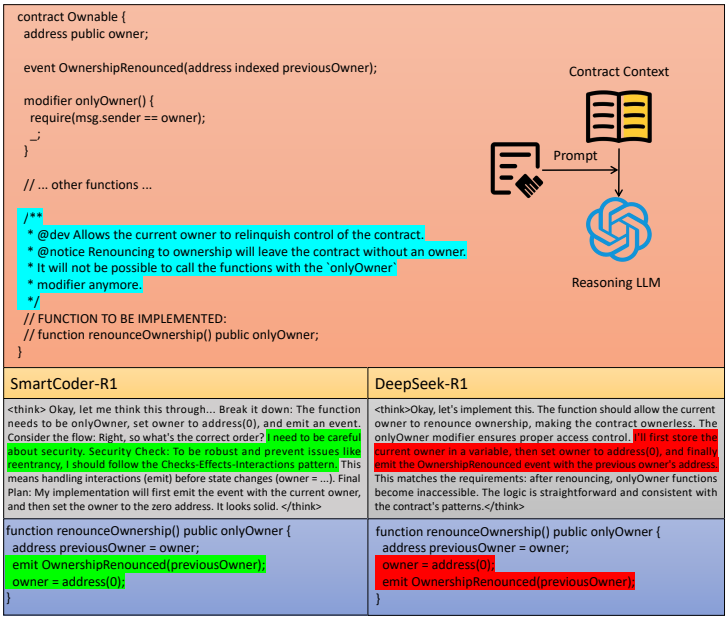


Fig. 4. Case study comparing the reasoning process and implementation order of two LLMs (SmartCoder-R1 and DeepSeek-R1) on the renounceOwnership function in smart contracts.

4) RQ4: We demonstrate reasoning and implementation differences between two LLMs through a comparative case study. As shown in Fig. 4, we examine how SmartCoder-R1 and DeepSeek-R1 (best baseline) implement the `renounceOwnership` function in an `Ownable` contract. Both generate a function allowing the owner to relinquish control. SmartCoder-R1 produces correct code while explicitly reasoning about security best practices, following the Checks-Effects-Interactions (CEI) pattern to prevent reentrancy by emitting events before state changes. In contrast, DeepSeek-R1 focuses on functional correctness without considering security nuances, changing state before emitting events, a potentially less secure approach. This highlights that while both achieve functional correctness, SmartCoder-R1 demonstrates deeper security understanding, proactively addressing risks beyond immediate requirements, whereas DeepSeek-R1 may overlook best practices crucial in real-world blockchain environments.

Answer to RQ4: SmartCoder-R1 applies security best practices like the CEI pattern, reducing vulnerability risks. In contrast, DeepSeek-R1 gives a solution without security awareness, underscoring the value of security-focused reasoning.

5) RQ5: To answer RQ5, we analyzed residual vulnerabilities in SmartCoder-R1 generated contracts. As shown in Table 6, among 756 test samples, only 57 contained vulnerabilities totaling

Table 6. Key Security Issues in Smart Contract Code Generated by SmartCoder-R1.

Security Issue	Count	Percentage (%)	Risk
Reentrancy Vulnerabilities	48	32.65	High
Array Bounds Unchecked	21	14.29	Med
Access Control Missing	18	12.24	High
State Validation Missing	15	10.20	Med
Integer Overflow/Underflow	13	8.84	Med
Improper Error Handling	10	6.80	Med
Timestamp Dependence	6	4.08	Med
Gas Limit DoS Risk	5	3.40	Low
Function Visibility Issues	4	2.72	Low
tx.origin Authentication	3	2.04	High
Selfdestruct Usage	2	1.36	High
Delegatecall Context Risk	2	1.36	High
Total	**147**	**100.00**	–

147 issues. Reentrancy was most common (32.65%, 48 instances), followed by unchecked array bounds (14.29%) and missing access controls (12.24%). Our risk classification reflects potential impact: High-risk vulnerabilities cause direct financial loss or control loss; Medium-risk ones cause functional anomalies or partial threats; Low-risk issues affect usability or minor security. These stem from occasional oversight of subtle security details in highly complex business logic, such as failing to anticipate all interaction risks in intricate state changes, rather than general security awareness failure. This analysis guides future improvements: enhancing training data with complex security counterexamples, integrating advanced static analysis into S-GRPO rewards, and refining the reasoning framework to eliminate these high-level vulnerabilities at their source.

Answer to RQ5: Analysis shows remaining vulnerabilities, mainly reentrancy (32.65%), occur in complex edge cases, not from a general security flaw. Future work will target these cases by enhancing training data with security counterexamples, integrating static analysis tools, and refining the security reasoning framework.

5 Related Work

5.1 Smart Contract Auditing and Generation

Smart Contract Auditing. Smart contract vulnerability detection has evolved from early program analysis techniques such as symbolic execution (Oyente [24]), pattern matching (Mythril [28], SmartCheck [45]), and formal verification (Securify [46]), to data-driven approaches including deep neural networks [33] and methods that use Graph Neural Networks (GNNs) [58, 68–70] to detect smart contract vulnerabilities [20, 72]. Recent advances employ pre-trained models (Peculiar [53]), prompt-tuning [62], and cross-modality learning [34]. The latest frontier leverages LLMs, with studies evaluating them on real-world datasets [3, 6], exploring reasoning capabilities [12], and developing tools like GPTScan [42], SOChecker [4], iAudit [26], and Smart-LLaMA [59, 61].

Smart Contract Generation. Recent work on LLM-based smart contract generation includes Storhaug et al.'s vulnerability tagging and constrained decoding [41], Wang et al.'s CodeBC with three-stage fine-tuning [47], and Alam et al.'s static analysis with multi-round masked prompting [1].

Despite these advances, most lack in-depth modeling of explainability and developer preferences. We propose SmartCoder-R1, a three-stage training paradigm integrating continual pre-training, long chain-of-thought supervised fine-tuning, and security-aware group relative policy optimization to enhance security, explainability, and usability.

5.2 Reinforcement Learning in LLM Reasoning

Reinforcement learning (RL) is a rapidly advancing frontier for LLM reasoning [48, 51]. A key approach is outcome-supervised RL, where models like DeepSeek-R1 are rewarded for the correctness of the final result, fostering complex reasoning without step-by-step supervision [9]. It has proven effective across diverse domains, including programming [8, 25, 49], finance [19], vision [13, 54], user interface automation [23], 3D spatial reasoning [31] and leveraging external tools [15, 16]. Inspired by these advancements, we adapt outcome-supervised RL to generate secure and functional smart contracts.

6 Threats to Validity

Internal Validity: The S-GRPO reward mechanism effectively addresses core objectives of compilation, security, and format compliance, but may not fully capture all code quality nuances such as conciseness or stylistic elegance, occasionally yielding correct but verbose outputs. Our parameter sensitivity analysis confirms optimal multi-objective balance, and training data underwent rigorous multi-stage validation (automated checks, expert review, and consensus resolution), leaving reward function refinement as a direction for future work.

External Validity: Two generalizability constraints apply. First, despite covering multiple Solidity versions and diverse contract categories, our expert-curated datasets represent a subset of the evolving smart contract landscape, and performance on emerging architectures or underrepresented domains warrants further validation. Second, as an Ethereum- and Solidity-specific model, applicability to other blockchain languages requires separate evaluation, and the knowledge cutoff introduces uncertainty for future vulnerability types.

Construct Validity: By focusing on compilability, security, and functional correctness, dimensions such as maintainability, gas efficiency, and emerging best practices remain underrepresented. The Likert scale evaluation for reasoning chains, despite high inter-rater reliability, retains inherent subjectivity that may not fully reflect all explanation quality aspects valued by developers.

7 Conclusion

In this paper, we introduced SmartCoder-R1, a novel framework designed to generate secure and explainable smart contracts. Our approach uniquely integrates a three-stage pipeline: Continual Pre-Training to build domain expertise, Long Chain-of-Thought Supervised Fine-Tuning to instill structured security reasoning, and Security-Aware Group Relative Policy Optimization to align the model with security standards. Extensive experiments demonstrate that SmartCoder-R1 significantly outperforms state-of-the-art baselines in generating code that is not only functionally correct and compilable but also substantially more secure and interpretable.

8 Data Availability

All data and source code that can be publicly released are available at https://zenodo.org/records/19691563

Acknowledgements

This work was supported by the National Key Research and Development Program of China (No.2023YFB3307203).

References

[1] Md Tauseef Alam, Sorbajit Goswami, Khushi Singh, Raju Halder, Abyayananda Maiti, and Soumyadip Banerjee. 2025. SolGen: Secure Smart Contract Code Generation Using Large Language Models Via Masked Prompting. In *Proceedings of the 18th Innovations in Software Engineering Conference (ISEC '25)*. Association for Computing Machinery, New York, NY, USA, Article 13, 11 pages. doi:10.1145/3717383.3717394

[2] Maher Alharby and Aad van Moorsel. 2017. Blockchain-based Smart Contracts: A Systematic Mapping Study. In *Fourth International Conference on Computer Science and Information Technology (CSIT-2017)*. doi:10.5121/csit.2017.71011

[3] Chong Chen, Jianzhong Su, Jiachi Chen, Yanlin Wang, Tingting Bi, Jianxing Yu, Yanli Wang, Xingwei Lin, Ting Chen, and Zibin Zheng. 2025. When ChatGPT Meets Smart Contract Vulnerability Detection: How Far Are We? *ACM Trans. Softw. Eng. Methodol.* 34, 4, Article 100 (April 2025), 30 pages. doi:10.1145/3702973

[4] Jiachi Chen, Chong Chen, Jiang Hu, John Grundy, Yanlin Wang, Ting Chen, and Zibin Zheng. 2024. Identifying Smart Contract Security Issues in Code Snippets from Stack Overflow. 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, 1198–1210. doi:10.1145/3650212.3680353

[5] Shiqi Cheng, Chenjie Shen, Li Yang, Lei Yu, Fengjun Zhang, and Chun Zuo. 2025. AUVANA: An Efficient and Automatic Approach to Variable Rename Refactoring via Large Pre-trained Language Model. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 288–299. doi:10.1109/ISSRE66568.2025.00038

[6] Isaac David, Liyi Zhou, Kaihua Qin, Dawn Song, Lorenzo Cavallaro, and Arthur Gervais. 2023. Do you still need a manual smart contract audit? *arXiv preprint arXiv:2306.12338* (2023). doi:10.48550/arXiv.2306.12338

[7] Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. 2017. The DAO hacked. *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you* (2017), 67–78.

[8] Lishui Fan, Yu Zhang, Mouxiang Chen, and Zhongxin Liu. 2025. ReCode: Reinforcing Code Generation with Reasoning-Process Rewards. *arXiv preprint arXiv:2508.05170* (Aug. 2025). arXiv:2508.05170 [cs.SE] doi:10.48550/arXiv.2508.05170

[9] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. *Nature* 645 (2025), 633–638. doi:10.1038/s41586-025-09422-z

[10] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, Y.K. Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196* (Jan. 2024). arXiv:2401.14196 [cs.SE] doi:10.48550/arXiv.2401.14196

[11] Tharaka Hewa, Mika Ylianttila, and Madhusanka Liyanage. 2021. Survey on blockchain based smart contracts: Applications, opportunities and challenges. *Journal of Network and Computer Applications* 177 (2021), 102857. doi:10.1016/j.jnca.2020.102857

[12] Sihao Hu, Tiansheng Huang, Fatih İlhan, Selim Furkan Tekin, and Ling Liu. 2023. Large Language Model-Powered Smart Contract Vulnerability Detection: New Perspectives. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. 297–306. doi:10.1109/TPS-ISA58951.2023.00044

[13] Wenxuan Huang, Bohan Jia, Zijie Zhai, Shaosheng Cao, Zheyu Ye, Fei Zhao, Zhe Xu, Xu Tang, Yao Hu, and Shaohui Lin. 2025. Vision-R1: Incentivizing Reasoning Capability in Multimodal Large Language Models. *arXiv preprint arXiv:2503.06749* (March 2025). arXiv:2503.06749 [cs.CV] doi:10.48550/arXiv.2503.06749

[14] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2.5-Coder Technical Report. *arXiv preprint arXiv:2409.12186* (Sept. 2024). arXiv:2409.12186 [cs.CL] doi:10.48550/arXiv.2409.12186

[15] Pengcheng Jiang, Jiacheng Lin, Lang Cao, Runchu Tian, SeongKu Kang, Zifeng Wang, Jimeng Sun, and Jiawei Han. 2025. DeepRetrieval: Hacking Real Search Engines and Retrievers with Large Language Models via Reinforcement Learning. In *Second Conference on Language Modeling*. https://openreview.net/forum?id=u9JXu4L17I

[16] Bowen Jin, Hansi Zeng, Zhenrui Yue, Jinsung Yoon, Sercan O Arik, Dong Wang, Hamed Zamani, and Jiawei Han. 2025. Search-R1: Training LLMs to Reason and Leverage Search Engines with Reinforcement Learning. In *Second Conference on Language Modeling*. https://openreview.net/forum?id=Rwhi91ideu

[17] Ankur Joshi, Saket Kale, Satish Chandel, and D Kumar Pal. 2015. Likert scale: Explored and explained. *British journal of applied science & technology* 7, 4 (2015), 396–403. doi:10.9734/BJAST/2015/14975

[18] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. DeepSeek-V3 Technical Report. arXiv:2412.19437 [cs.CL] doi:10.48550/arXiv.2412.19437

[19] Zhaowei Liu, Xin Guo, Zhi Yang, Fangqi Lou, Lingfeng Zeng, Jinyi Niu, Mengping Li, Qi Qi, Zhiqiang Liu, Yiyang Han, Dongpo Cheng, Ronghao Chen, Huacan Wang, Xingdong Feng, Huixia Judy Wang, Chengchun Shi, and Liwen Zhang. 2025. Fin-R1: A Large Language Model for Financial Reasoning through Reinforcement Learning. *arXiv preprint arXiv:2503.16252* (March 2025). arXiv:2503.16252 [cs.CL] doi:10.48550/arXiv.2503.16252

[20] Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: from pure neural network to interpretable graph feature and expert pattern fusion. *arXiv preprint*

arXiv:2106.09282 (2021). doi:10.48550/arXiv.2106.09282

[21] Ilya Loshchilov and Frank Hutter. 2017. Decoupled Weight Decay Regularization. *arXiv preprint arXiv:1711.05101* (Nov. 2017). arXiv:1711.05101 [cs.LG] doi:10.48550/arXiv.1711.05101

[22] Junyi Lu, Lei Yu, Xiaojia Li, Li Yang, and Chun Zuo. 2023. LLaMA-Reviewer: Advancing Code Review Automation with Large Language Models through Parameter-Efficient Fine-Tuning. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 647–658. doi:10.1109/ISSRE59848.2023.00026

[23] Zhengxi Lu, Yuxiang Chai, Yaxuan Guo, Xi Yin, Liang Liu, Hao Wang, Han Xiao, Shuai Ren, Pengxiang Zhao, Guangyi Liu, Guanjing Xiong, and Hongsheng Li. 2026. UI-R1: Enhancing Efficient Action Prediction of GUI Agents by Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 17608–17616. doi:10.1609/aaai.v40i21.38816

[24] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 254–269. doi:10.1145/2976749.2978309

[25] Peixian Ma, Xialie Zhuang, Chengjin Xu, Xuhui Jiang, Ran Chen, and Jian Guo. 2025. SQL-R1: Training Natural Language to SQL Reasoning Model By Reinforcement Learning. In *Advances in Neural Information Processing Systems*, D. Belgrave, C. Zhang, H. Lin, R. Pascanu, P. Koniusz, M. Ghassemi, and N. Chen (Eds.), Vol. 38. Curran Associates, Inc., 174505–174537. https://proceedings.neurips.cc/paper_files/paper/2025/file/ff00aae6ce2b5e519ca4cf769f0c4a47-Paper-Conference.pdf

[26] Wei Ma, Daoyuan Wu, Yuqiang Sun, Tianwen Wang, Shangqing Liu, Jian Zhang, Yue Xue, and Yang Liu. 2025. Combining Fine-Tuning and LLM-Based Agents for Intuitive Smart Contract Auditing with Justifications. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1742–1754. doi:10.1109/ICSE55347.2025.00027

[27] Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. 2019. Understanding a Revolutionary and Flawed Grand Experiment in Blockchain: The DAO Attack. *Journal of Cases on Information Technology (JCIT)* 21, 1 (2019), 19–32. doi:10.4018/JCIT.2019010102

[28] B Mueller. 2017. Mythril-Reversing and bug hunting framework for the Ethereum blockchain. https://pypi.org/project/mythril/0.8.2

[29] Emanuele Antonio Napoli, Fadi Barbàra, Valentina Gatteschi, and Claudio Schifanella. 2024. Leveraging Large Language Models for Automatic Smart Contract Generation. In *2024 IEEE 48th Annual Computers, Software, and Applications Conference (COMPSAC)*. 701–710. doi:10.1109/COMPSAC61105.2024.00100

[30] OpenAI. 2025. GPT-4.1. https://platform.openai.com/docs/models/gpt-4.1.

[31] Zhenyu Pan and Han Liu. 2025. MetaSpatial: Reinforcing 3D Spatial Reasoning in VLMs for the Metaverse. *arXiv preprint arXiv:2503.18470* (March 2025). arXiv:2503.18470 [cs.CV] doi:10.48550/arXiv.2503.18470

[32] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a Method for Automatic Evaluation of Machine Translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*. 311–318. https://aclanthology.org/P02-1040.pdf

[33] Peng Qian, Zhenguang Liu, Qinming He, Roger Zimmermann, and Xun Wang. 2020. Towards Automated Reentrancy Detection for Smart Contracts Based on Sequential Models. *IEEE Access* 8 (2020), 19685–19695. doi:10.1109/ACCESS.2020.2969429

[34] Peng Qian, Zhenguang Liu, Yifang Yin, and Qinming He. 2023. Cross-Modality Mutual Learning for Enhancing Smart Contract Vulnerability Detection on Bytecode. In *Proceedings of the ACM Web Conference 2023* (Austin, TX, USA) *(WWW '23)*. Association for Computing Machinery, New York, NY, USA, 2220–2229. doi:10.1145/3543507.3583367

[35] Qwen. 2025. QwQ-32B. https://huggingface.co/Qwen/QwQ-32B.

[36] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. DeepSpeed: System Optimizations Enable Training Deep Learning Models with Over 100 Billion Parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 3505–3506. doi:10.1145/3394486.3406703

[37] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. CodeBLEU: A Method for Automatic Evaluation of Code Synthesis. *arXiv preprint arXiv:2009.10297* (Sept. 2020). arXiv:2009.10297 [cs.SE] doi:10.48550/arXiv.2009.10297

[38] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv preprint arXiv:2308.12950* (Aug. 2023). arXiv:2308.12950 [cs.CL] doi:10.48550/arXiv.2308.12950

[39] Chenjie Shen, Jie Zhu, Lei Yu, Li Yang, and Chun Zuo. 2024. Dependency-Aware Method Naming Framework with Generative Adversarial Sampling. In *2024 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN60899.2024.10651109

[40] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2025. HybridFlow: A Flexible and Efficient RLHF Framework. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1279–1297. doi:10.1145/3689031.3696075

[41] 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)*. 683–693. doi:10.1109/ISSRE59848.2023.00035

[42] Yuqiang Sun, Daoyuan Wu, Yue Xue, Han Liu, Haijun Wang, Zhengzi Xu, Xiaofei Xie, and Yang Liu. 2024. GPTScan: Detecting Logic Vulnerabilities in Smart Contracts by Combining GPT with Program Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 166, 13 pages. doi:10.1145/3597503.3639117

[43] Melanie Swan. 2015. *Blockchain: Blueprint for a New Economy*. O'Reilly Media, Inc. 152 pages.

[44] Jiayue Tang, Li Yang, Lei Yu, Junyi Lu, Zhirong Huang, Fengjun Zhang, and Chun Zuo. 2025. Breaking Task Isolation: Enhancing Code Review Automation with Mixture-of-Experts Large Language Models. In *2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE)*. 227–238. doi:10.1109/ISSRE66568.2025.00033

[45] Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: static analysis of ethereum smart contracts. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (Gothenburg, Sweden) *(WETSEB '18)*. Association for Computing Machinery, New York, NY, USA, 9–16. doi:10.1145/3194113.3194115

[46] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 67–82. doi:10.1145/3243734.3243780

[47] Lingxiang Wang, Hainan Zhang, Qinnan Zhang, Ziwei Wang, Hongwei Zheng, Jin Dong, and Zhiming Zheng. 2025. CodeBC: A More Secure Large Language Model for Smart Contract Code Generation in Blockchain. *arXiv preprint arXiv:2504.21043* (April 2025). arXiv:2504.21043 [cs.CR] doi:10.48550/arXiv.2504.21043

[48] Qineng Wang, Zihao Wang, Ying Su, Hanghang Tong, and Yangqiu Song. 2024. Rethinking the Bounds of LLM Reasoning: Are Multi-Agent Discussions the Key?. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 6106–6131. doi:10.18653/v1/2024.acl-long.331

[49] Sijie Wang, Quanjiang Guo, Kai Zhao, Yawei Zhang, Xin Li, Xiang Li, Siqi Li, Rui She, Shangshu Yu, and Wee Peng Tay. 2025. CodeBoost: Boosting Code LLMs by Squeezing Knowledge from Code Snippets with RL. *arXiv preprint arXiv:2508.05242* (Aug. 2025). arXiv:2508.05242 [cs.CL] doi:10.48550/arXiv.2508.05242

[50] Zengzhi Wang, Fan Zhou, Xuefeng Li, and Pengfei Liu. 2025. OctoThinker: Mid-Training Incentivizes Reinforcement Learning Scaling. In *2nd AI for Math Workshop @ ICML 2025*. https://openreview.net/forum?id=chCeUHjLzs

[51] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems*, S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (Eds.), Vol. 35. Curran Associates, Inc., 24824–24837. https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf

[52] Gavin Wood et al. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper* 151, 2014 (2014), 1–32. https://ethereum.github.io/yellowpaper/paper.pdf

[53] Hongjun Wu, Zhuo Zhang, Shangwen Wang, Yan Lei, Bo Lin, Yihao Qin, Haoyu Zhang, and Xiaoguang Mao. 2021. Peculiar: Smart Contract Vulnerability Detection Based on Crucial Data Flow Graph and Pre-training Techniques. In *2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE)*. 378–389. doi:10.1109/ISSRE52982.2021.00047

[54] Jiaer Xia, Yuhang Zang, Peng Gao, Sharon Li, and Kaiyang Zhou. 2025. Visionary-R1: Mitigating Shortcuts in Visual Reasoning with Reinforcement Learning. *arXiv preprint arXiv:2505.14677* (May 2025). arXiv:2505.14677 [cs.CV] doi:10.48550/arXiv.2505.14677

[55] Tian Xie, Zitian Gao, Qingnan Ren, Haoming Luo, Yuqian Hong, Bryan Dai, Joey Zhou, Kai Qiu, Zhirong Wu, and Chong Luo. 2025. Logic-RL: Unleashing LLM Reasoning with Rule-Based Reinforcement Learning. *arXiv preprint arXiv:2502.14768* (Feb. 2025). arXiv:2502.14768 [cs.CL] doi:10.48550/arXiv.2502.14768

[56] An Yang, Anfeng Li, Baosong Yang, et al. 2025. Qwen3 Technical Report. arXiv:2505.09388 [cs.CL] doi:10.48550/arXiv.2505.09388

[57] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. 2024. Qwen2.5 Technical Report. arXiv:2412.15115 [cs.CL] doi:10.48550/arXiv.2412.15115

[58] Yuanzhe Yang, Li Yang, Lingwei Li, Xiaoxiao Ma, Lei Yu, and Chun Zuo. 2023. DccGraph: Detecting Criminal Communities with Augmented Criminal Network Construction and Graph Neural Network. In *2023 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN54540.2023.10191121

[59] Lei Yu, Shiqi Chen, Hang Yuan, Peng Wang, Zhirong Huang, Jingyuan Zhang, Chenjie Shen, Fengjun Zhang, Li Yang, and Jiajia Ma. 2024. Smart-LLaMA: Two-Stage Post-Training of Large Language Models for Smart Contract Vulnerability Detection and Explanation. *arXiv preprint arXiv:2411.06221* (2024). doi:10.48550/arXiv.2411.06221

[60] Lei Yu, Shiqi Cheng, Zhirong Huang, Jingyuan Zhang, Chenjie Shen, Junyi Lu, Li Yang, Fengjun Zhang, and Jiajia Ma. 2025. SAEL: Leveraging Large Language Models with Adaptive Mixture-of-Experts for Smart Contract Vulnerability Detection. In *2025 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 61–72. doi:10.1109/ICSME64153.2025.00016

[61] 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

[62] Lei Yu, Junyi Lu, Xianglong Liu, Li Yang, Fengjun Zhang, and Jiajia Ma. 2023. PSCVFinder: A Prompt-Tuning Based Framework for Smart Contract Vulnerability Detection. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*. 556–567. doi:10.1109/ISSRE59848.2023.00030

[63] Lei Yu, Peng Wang, Jingyuan Zhang, Xin Wang, Jia Xu, Li Yang, Changzhi Deng, Jiajia Ma, and Fengjun Zhang. 2026. SQL-Commenter: Aligning Large Language Models for SQL Comment Generation with Direct Preference Optimization. *arXiv preprint arXiv:2603.18606* (2026). doi:10.48550/arXiv.2603.18606

[64] Lei Yu, Fengjun Zhang, Jiajia Ma, Li Yang, Yuanzhe Yang, and Wei Jia. 2023. Who Are the Money Launderers? Money Laundering Detection on Blockchain via Mutual Learning-Based Graph Neural Network. In *2023 International Joint Conference on Neural Networks (IJCNN)*. 1–8. doi:10.1109/IJCNN54540.2023.10191217

[65] Hang Yuan, Xizhi Hou, Lei Yu, Li Yang, Jiayue Tang, Jiadong Xu, Yifei Liu, Fengjun Zhang, and Chun Zuo. 2025. Leveraging Mixture-of-Experts Framework for Smart Contract Vulnerability Repair with Large Language Model. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1667–1679. doi:10.1109/ASE63991.2025.00140

[66] Hang Yuan, Lei Yu, Zhirong Huang, Jingyuan Zhang, Junyi Lu, Shiqi Cheng, Li Yang, Fengjun Zhang, Jiajia Ma, and Chun Zuo. 2025. Mos: Towards effective smart contract vulnerability detection through mixture-of-experts tuning of large language models. *arXiv preprint arXiv:2504.12234* (2025). doi:10.48550/arXiv.2504.12234

[67] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. 2024. Swe-bench-java: A github issue resolving benchmark for java. *arXiv preprint arXiv:2408.14354* (2024). doi:10.48550/arXiv.2408.14354

[68] Jingyuan Zhang, Xin Wang, Lei Yu, Zhirong Huang, Li Yang, and Fengjun Zhang. 2025. Restricted Global-Aware Graph Filters Bridging GNNs and Transformer for Node Classification. In *Advances in Neural Information Processing Systems*, Vol. 38. Curran Associates, Inc., 162403–162429. https://proceedings.neurips.cc/paper_files/paper/2025/file/edbcb7583fd8921dad78adecfe06a99b-Paper-Conference.pdf

[69] Jingyuan Zhang, Xin Wang, Lei Yu, Li Yang, and Fengjun Zhang. 2026. Binary Message Passing for Generalizable Semi-Supervised Graph Anomaly Detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 40. 16334–16342. doi:10.1609/aaai.v40i19.38671

[70] Jingyuan Zhang, Lei Yu, Zhirong Huang, Li Yang, and Fengjun Zhang. 2025. Topology Augmented Multi-Band and Multi-Scale Filtering for Graph Anomaly Detection. *ACM Trans. Knowl. Discov. Data* 19, 8, Article 151 (Sept. 2025), 27 pages. doi:10.1145/3748727

[71] Yaowei Zheng, Richong Zhang, Junhao Zhang, Yanhan Ye, and Zheyan Luo. 2024. LlamaFactory: Unified Efficient Fine-Tuning of 100+ Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, Yixin Cao, Yang Feng, and Deyi Xiong (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 400–410. doi:10.18653/v1/2024.acl-demos.38

[72] Yuan Zhuang, Zhenguang Liu, Peng Qian, Qi Liu, Xiang Wang, and Qinming He. 2021. Smart contract vulnerability detection using graph neural networks. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (Yokohama, Yokohama, Japan) *(IJCAI'20)*. Article 454, 8 pages.

[73] Weiqin Zou, David Lo, Pavneet Singh Kochhar, Xuan-Bach Dinh Le, Xin Xia, Yang Feng, Zhenyu Chen, and Baowen Xu. 2021. Smart Contract Development: Challenges and Opportunities. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2084–2106. doi:10.1109/TSE.2019.2942301

Received 2025-09-04; accepted 2026-03-24