

1 Securing Multiple Smart Contract Languages: A Unified, Agentic 2 Framework for Vulnerability Repair in Solidity and Move

3 Anonymous Author(s)

4 Abstract

5 The rapid growth of the blockchain ecosystem and the increasing
6 value locked in smart contracts necessitate robust security measures.
7 While languages like Solidity and Move aim to improve smart
8 contract security, vulnerabilities persist. This paper presents Smartify,
9 a novel multi-agent framework leveraging Large Language Models
10 (LLMs) to automatically detect and repair vulnerabilities in
11 Solidity and Move smart contracts. Unlike traditional methods that
12 rely solely on vast pre-training datasets, Smartify employs a team of
13 specialized agents working on different specially fine-tuned LLMs
14 to analyze code based on the underlying programming concepts
15 and language-specific security principles. We evaluated Smartify on
16 a dataset for Solidity and a curated dataset for Move, demonstrating
17 its effectiveness in fixing a wide range of vulnerabilities. Our
18 experimental results show that Smartify (Gemma2+Codegemma) achieves
19 state-of-the-art performance, surpassing existing LLMs and even en-
20 hancing the capabilities of general-purpose models, such as Llama
21 3.1. Notably, Smartify can incorporate language-specific knowledge,
22 such as the nuances of Move, without requiring massive language-
23 specific pretraining datasets. This work offers a detailed analysis of
24 the performance of various LLMs on smart contract repair, high-
25 lighting the strengths of our multi-agent approach and providing
26 a blueprint for developing more secure and reliable decentralized
27 applications in the growing blockchain landscape. We also provide
28 a detailed description to extend the proposed technology to other
29 similar use cases.

34 ACM Reference Format:

35 Anonymous Author(s). 2025. Securing Multiple Smart Contract Languages:
36 A Unified, Agentic Framework for Vulnerability Repair in Solidity and Move.
37 In . ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/nnnnnnn>.
38 nnnnnnnn

39 1 Introduction

40 Smart contracts, self-executing agreements with terms directly
41 written into code, have emerged as a cornerstone of blockchain
42 technology [28, 31]. Their ability to automate transactions and
43 eliminate intermediaries has led to widespread adoption in various
44 sectors, including finance, supply chain management, and
45 healthcare [21, 22, 39]. However, the increasing complexity of smart
46 contracts has given rise to a growing concern: security vulnerabili-
47 ties [35]. These vulnerabilities, often stemming from coding errors

48
49 Permission to make digital or hard copies of all or part of this work for personal or
50 classroom use is granted without fee provided that copies are not made or distributed
51 for profit or commercial advantage and that copies bear this notice and the full citation
52 on the first page. Copyrights for components of this work owned by others than the
53 author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or
54 republish, to post on servers or to redistribute to lists, requires prior specific permission
55 and/or a fee. Request permissions from permissions@acm.org.

56 Conference'17, July 2017, Washington, DC, USA

57 © 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

58 ACM ISBN 978-x-xxxx-xxxx-x/YY/MM

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

59 or design flaws, can be exploited by malicious actors, leading to significant financial losses and damage to the reputation of blockchain projects.

60 The financial implications of smart contract vulnerabilities are substantial. Reports indicate that cumulative losses from attacks against Ethereum smart contracts alone have exceeded USD 3.1 billion by 2023 [24]. In the DeFi space, an estimated \$9.04 billion has been stolen due to vulnerabilities [38]. Notable incidents like the DAO hack of 2016, resulting in a \$55 million loss [30], and the Poly Network hack in 2021, where over \$600 million was stolen [4], underscore the critical need for robust security measures.

61 Traditional security auditing methods, although essential, often face limitations in terms of accuracy and scalability. This has spurred the exploration of automated techniques for vulnerability detection [18, 37] and repair, with Large Language Models (LLMs) emerging as a promising solution [20]. LLMs trained on vast datasets of code can learn to understand and generate code that adheres to specific programming paradigms and best practices. However, most of the tools available for smart contract safety are language-specific, especially Solidity [9]. For other languages, existing tools often require scanning of compiled bytecode [33].

62 Apart from Solidity, Move [5] has gained significant traction due to its strong focus on security. Its cutting-edge features, including a custom data type for secure operations, robust access controls via Move modules, and unique memory safety features [7], have been particularly noteworthy. Moreover, the Move Prover, a native security framework, provides an additional layer of protection [11]. Notably, several prominent blockchain platforms, such as Starcoin [3], Aptos [10], and Sui [6], have already adopted Move.

63 However, despite its promising architecture, the real-world security performance of Move modules remains largely untested. Unlike Solidity-based smart contracts, which have been extensively studied through empirical research and surveys, there is a scarcity of research focused specifically on Move modules. Although some methodologies have been proposed for identifying defects in Move modules or conducting formal verification [23, 29], and empirical analysis [33], a significant knowledge gap persists. Specifically, large-scale investigations into the frequency of defects in real-world Move modules and identifying and repairing potential vulnerabilities are lacking, highlighting the need for further research in this area.

64 This paper introduces Smartify, a framework that moves beyond naive LLM application by proposing a novel paradigm for structured, automated repair. We introduce Smartify not merely as a "multi-agent framework" but as a **structured, collaborative system that mimics the workflow of an expert human security audit team**. This contrasts with existing LLM-powered tools like ContractTinker, which utilize a linear Chain-of-Thought (CoT) reasoning process to break down the repair task. Smartify's novelty

lies in its process-centric design, where specialized agents (Auditor, Architect, Code Generator, Refiner, Validator) assume distinct roles within a delegative and iterative workflow. The Architect agent, for instance, does not simply reason about the next step; it formulates a comprehensive repair strategy that is then executed by other agents. Crucially, the Refiner-Validator duo establishes a feedback loop for iterative quality assurance, a feature essential for generating trustworthy patches.

In this paper, our aim is to answer the following research questions related to software engineering using AI agents and the landscape of complex smart contract reasoning.

- **RQ1:** Do the present state-of-the-art LLMs can explain a Smart Contract code correctly?
- **RQ2:** Can they detect and explain bad coding practices or specific mistakes leading to bugs or vulnerabilities in a smart contract code?
- **RQ3:** Can we encode programming language-specific knowledge to train the LLMs to understand unsafe and buggy codes in detail enough to repair them?
- **RQ4:** Can LLMs repair the bugs and fix the vulnerability?
- **RQ5:** Does the proposed post-training framework be generalizable to a larger set of pre-trained LLMs?

The main contributions of our work are the following:

- We introduce a novel **role-based multi-agent architecture** for smart contract repair that models the collaborative workflow of a security audit team, enhancing structured reasoning over monolithic LLM approaches.
- We propose a rigorous methodology combining **specialized fine-tuned models for deep vulnerability analysis**, **Retrieval-Augmented Generation (RAG)** for language-specific context, and an **iterative self-refinement loop** for ensuring patch quality.
- We conduct the first **multi-faceted empirical evaluation** of an LLM-based repair tool for both Solidity and Move, using metrics that assess not only correctness but also **exploit mitigation effectiveness**, **semantic preservation**, and **code quality**.
- We present a comprehensive **ablation study** that systematically dissects our framework to quantify the distinct contribution of each architectural component to the overall repair performance.

2 Related Work

This section provides a critical review of the existing landscape in smart contract security, positioning Smartify relative to traditional analysis techniques and emerging LLM-based repair methodologies.

2.1 Traditional Smart Contract Security Analysis

Automated security analysis for smart contracts has traditionally been dominated by static, dynamic, and formal verification techniques. Each approach offers distinct advantages but also possesses inherent limitations, particularly when confronted with complex, logic-based vulnerabilities.

2.2 Smart Contract Security Auditing

Various tools and techniques have been developed for detecting vulnerabilities in smart contracts:

Static Analysis Tools: Tools like Mythril [27] and Slither [12] analyze contract source code to identify potential vulnerabilities.

They perform symbolic execution and taint analysis to detect patterns associated with common vulnerabilities.

Dynamic Analysis Tools: Tools like Manticore [26] and Echidna [13] execute contracts with various inputs to uncover runtime errors. They use fuzzing and symbolic execution techniques to explore different execution paths and identify potential issues.

Formal Verification: This approach uses mathematical techniques to rigorously prove the correctness of a contract's code against a formal specification. Tools like KEVM [15] and CertiK's DeepSEA have been developed for formal verification of smart contracts [40].

While these tools are valuable, they often have limitations in accuracy, scalability, and the ability to handle the complexities of real-world smart contracts.

2.3 LLM-Powered Smart Contract Repair: A Taxonomy

The application of LLMs to automated program repair is a rapidly advancing field, with several distinct methodologies emerging.

2.3.1 Monolithic LLM Approaches Early efforts in this domain involved using large, general-purpose LLMs like GPT-3 with intricate, zero-shot or few-shot prompts to generate patches. While demonstrating feasibility, these approaches often lack the domain-specific knowledge required for the high-stakes environment of smart contracts. They are prone to generating syntactically correct but semantically flawed or insecure code, as they lack a deep, ingrained understanding of blockchain-specific security paradigms [7, 8].

2.3.2 Chain-of-Thought (CoT) and Static Analysis Integration To address the limitations of monolithic models, more structured approaches have been developed. A prominent example is **ContractTinker**, a tool designed for real-world smart contract repair [36]. ContractTinker employs a Chain-of-Thought (CoT) mechanism to guide an LLM through a sequence of reasoning steps: vulnerability localization, analysis, and patch generation. To ground the LLM's reasoning and mitigate hallucination, it integrates static analysis techniques, including dependency analysis and program slicing, to provide relevant context from audit reports and the source code itself.

While ContractTinker represents a significant advancement by imposing a logical structure on the repair process, its CoT approach remains a fundamentally linear and sequential reasoning pipeline. In contrast, Smartify's architecture is **delegative and iterative**. The Architect agent formulates a comprehensive, high-level plan, which is then delegated to specialized agents for execution. Furthermore, the explicit Refiner-Validator loop introduces a crucial mechanism for feedback and iterative quality improvement, a feature not explicitly detailed in the ContractTinker workflow. This architectural distinction moves beyond a simple chain of thought to a collaborative problem-solving process.

2.3.3 Multi-Agent Systems for Code Tasks The concept of using multiple LLM-based agents to collaborate on complex tasks has gained traction in the broader software engineering domain, with applications in areas like code translation and generation [17, 19]. These systems leverage the principle of specialization, assigning

233 different roles or sub-tasks to individual agents to achieve a more
 234 robust and accurate outcome than a single agent could.

235 While Smartify aligns with this general trend, its novelty lies
 236 in its **domain-specific agent roles tailored explicitly for the**
 237 **vulnerability repair workflow**. Instead of a generic "translator"
 238 or "coder" agent, Smartify's agents embody the distinct functions of
 239 a human security audit team: the Auditor for analysis, the Architect
 240 for strategic planning, the Code Generator for implementation,
 241 and the Refiner/Validator for quality assurance. This specialization
 242 allows for a more nuanced and effective approach to the highly
 243 specific and critical task of securing smart contracts.

244 Our proposed framework, **Smartify**, addresses these challenges
 245 by combining the strengths of specialized LLMs within a multi-
 246 agent architecture. It leverages language-specific fine-tuning, safety
 247 classifiers, and Retrieval-Augmented Generation (RAG) to enhance
 248 the accuracy and security of generated code repairs.

249 In the following sections, we detail the architecture of Smartify,
 250 describe the experimental setup, present the evaluation results, and
 251 discuss the implications of our findings for the future of smart
 252 contract security.

253 3 Data Collection and Analysis Methodology

254 This research employs a multi-faceted approach to investigate the
 255 security of smart contracts, focusing on both Solidity and Move
 256 programming languages. The methodology encompasses collecting
 257 and analyzing two distinct datasets: Solidity-based, Move-based
 258 source code. Each dataset serves a specific purpose in addressing
 259 the research questions and contributing to a comprehensive under-
 260 standing of smart contract vulnerabilities.

261 3.1 Importance of Dataset Categorization

262 For several reasons, categorizing the datasets based on program-
 263 ming language (Solidity and Move) and code representation is
 264 crucial. It allows for a focused analysis of language-specific vulnera-
 265 bilities and coding practices. As a more mature language, Solidity
 266 exhibits a different vulnerability landscape than the newer Move
 267 language. Examining them separately enables the identification of
 268 unique challenges and security considerations associated with each
 269 language.

270 3.2 Dataset Descriptions

271 3.2.1 *Solidity-based Dataset* This dataset comprises a collection
 272 of vulnerable Solidity smart contracts sourced from the "Not-
 273 So-Smart Contracts" repository curated by Trail of Bits [2]. This
 274 repository is renowned for its comprehensive set of contracts that
 275 intentionally exhibit a variety of common vulnerabilities. These vul-
 276 nerabilities were chosen for inclusion because of their prevalence
 277 in real-world decentralized applications and their representation of
 278 typical errors during smart contract development. The dataset con-
 279 tains 60 vulnerable contracts, encompassing 8 distinct vulnerability
 280 categories. Table 1 shows these categories' distribution.

281 3.2.2 *Move-Based Dataset (Source Code)* This dataset encompasses
 282 the source code of 92 real-world Move projects, comprising 652
 283 individual modules. These projects were part of Aptos [10], Sui [6],
 284 and Starcoin [3]. These projects span various application domains,
 285 as depicted in Table 2. The total number of Move projects is 92, and
 286 the total number of Move modules within these projects is 652.

287 **Table 1: Distribution of Vulnerabilities in the Solidity Dataset.**

Vulnerability Type	Number of Contracts	Percentage (%)
Reentrancy	15	25.0
Integer Overflow/Underflow	10	16.7
Denial of Service (DoS)	8	13.3
Access Control Issues	12	20.0
Uninitialized Storage Pointers	5	8.3
Tx.origin Misuse	3	5.0
Timestamp Dependency	4	6.7
Gas Limit and Out-of-Gas Vulnerabilities	3	5.0
Total	60	100.0

298 **Table 2: Distribution of Move Projects by Application Domain.**

Application Domain	Number of Projects	Percentage (%)
Decentralized Finance	41	44.6
Token	22	23.9
Bridge	18	19.6
Library	3	3.3
Infrastructure	3	3.3
Other	5	5.4
Total	92	100.0

299 For both the Move-based datasets, we utilize Song et al.'s [33]
 300 work to compare the vulnerability detection part. While the prior
 301 work is directed towards detection, the same dataset helps us com-
 302 pare Smartify's performance on both detection and repair.

303 3.3 Evaluation of Smartify

304 Smartify is designed with two core functionalities: detecting and
 305 repairing unsafe coding patterns in smart contracts. We utilize the
 306 previously described datasets to evaluate these capabilities rigor-
 307 ously, encompassing both Solidity and Move code. The evaluation
 308 process focuses on the complete output of Smartify rather than
 309 individual components, reflecting its nature as an integrated solu-
 310 tion for smart contract security. Performance is measured using the
 311 Pass@1 score.

312 3.4 Agent-Based Code Repair Process for Smart 313 Contracts

314 Our approach leverages a multi-agent system inspired by estab-
 315 lished software development methodologies but tailored explicitly
 316 for the automated repair of Solidity and Move smart contracts.
 317 This system employs five specialized agents: an *Auditor*, an *Architect*,
 318 a *Code Generator*, a *Refiner*, and a *Validator*. The process
 319 incorporates a self-refinement loop and a final validation step, en-
 320 suring high accuracy and security. Each agent plays a distinct role
 321 in a structured workflow, detailed below.

322 • **Auditor:** This agent is the cornerstone of security analysis.
 323 It is fine-tuned on a comprehensive corpus of Solidity and
 324 Move code documentation, encompassing syntax, semantics, and
 325 best practices. Furthermore, it is safety-aligned using a classifier
 326 adapted from Google's Responsible AI toolkit. This classifier has
 327 been meticulously modified to enforce language-specific rules
 328 and safe coding practices, effectively preventing the generation
 329 of unsafe or unsupported code constructs.

330 This alignment is of paramount importance. For Move, it ensures
 331 that the generated code strictly adheres to the conventions of the
 332 target blockchain (e.g., Sui or Aptos). It prevents the accidental
 333 introduction of elements from one Move variant into another
 334 or the inclusion of unsupported Rust paradigms. This is cru-
 335 cial because Move, derived from Rust, has unique features and
 336 limitations. For Solidity, it enforces established security best
 337 practices and prevents the generation of code patterns known to
 338 be vulnerable.

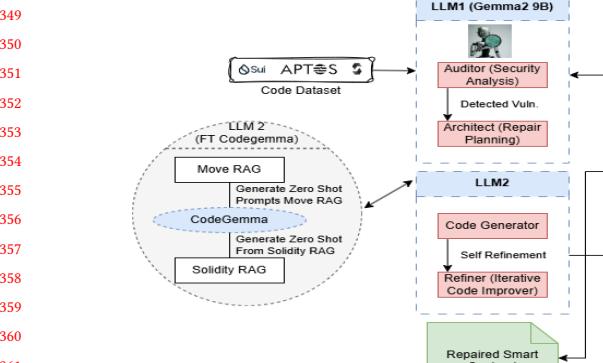


Figure 1: Agentic architecture of Smartify.

The Auditor's primary responsibility is to meticulously scan the input smart contract code (either Solidity or Move) to identify potential vulnerabilities and unsafe patterns. Its secondary, yet vital, role is to serve as the final *Validator* of the repaired code.

- Architect:** This agent receives the output from the *Auditor*, which includes a detailed report of identified vulnerabilities and unsafe code segments. The *Architect*'s role is to devise a high-level strategic plan for addressing these issues. This plan does not involve generating code directly. Instead, it outlines the necessary modifications, refactoring, and improvements to rectify the identified problems. This plan is a comprehensive blueprint for the *Code Generator*, guiding the code repair process.
- Code Generator:** This agent is a general-purpose code LLM. Its strength lies in its ability to leverage Retrieval-Augmented Generation (RAG) from two distinct data stores, one dedicated to Solidity and the other to Move. These data stores contain a collection of best practices and relevant documentation for the respective programming languages. Using the *Architect*'s plan as a guide, the *Code Generator* selects and adapts relevant examples from the appropriate RAG data store. This dynamic, context-aware retrieval of few-shot examples significantly enhances the *Code Generator*'s ability to produce accurate and secure code repairs. It ensures the generated code adheres to language-specific conventions and incorporates established best practices.
- Refiner:** This agent's role is to enhance the code quality produced by the *Code Generator*. It achieves this through iterative self-refinement, essentially acting as its critic. The *Refiner* uses the same underlying LLM as the *Code Generator* but with a different prompt that focuses on improving the code quality based on best practices and potential improvements that it might detect from a higher level.
- Validator:** This agent acts as a final checkpoint. After the refinement stage, it re-employs the *Auditor* agent to re-evaluate the code. The *Validator*'s objective is to ensure that all the previously identified vulnerabilities have been adequately addressed and that no new vulnerabilities have been introduced during the repair and refinement process.

4 Smartify System Architecture and Workflow

Smartify operates through a five-agent system designed for automated intelligent contract vulnerability detection and repair. The system functions as shown in Figure 1.

The Smartify system operates in a five-phase process to automatically repair smart contract code. Firstly, in the Input & Initial Audit phase, the smart contract code, written in either Solidity or Move, is fed into the system. The *Auditor*, an LLM based on Gemma2 9B, analyzes the code to detect potential vulnerabilities and produces a report detailing its findings. Secondly, during Repair Planning, the *Architect* receives this vulnerability report and formulates a high-level repair plan that outlines the necessary code modifications to address the identified issues. Thirdly, in Code Generation & Refinement, an LLM called *CodeGemma*, which has been fine-tuned for code generation and is equipped with Retrieval-Augmented Generation (RAG) capabilities, takes the lead. It utilizes separate Move RAG and Solidity RAG components to provide language-specific context. The *Code Generator*, part of *CodeGemma*, uses the repair plan to generate the modified code, selecting the appropriate RAG based on the input language and able to perform Solidity to Move translation when necessary.

Subsequently, a Self-refinement process is initiated, and the *Refiner* component iteratively improves the generated code's quality, readability, and efficiency. Fourthly, in the Validation phase, the *Validator* (the same agent as the *Auditor*) performs a final security audit on the refined code to ensure all identified vulnerabilities have been resolved. Finally, the system outputs the repaired smart contract code.

The process may iterate to step 3 or 4 if the *Validator* identifies any issues. Each step is vital in ensuring the smart contract code's accurate and secure repair. The workflow is designed to be efficient and effective, leveraging each agent's strengths to achieve the desired outcome.

4.1 Agent Prompting Strategy

The agents within Smartify are driven by carefully crafted prompts that guide their actions and ensure consistent performance. We employ a standardized prompt template adapted from established practices in LLM-based agent systems. The template is structured as follows:

Prompt Template

Role: You are a [role] specializing in [Solidity/Move] smart contracts.

Task: [task]

Instruction: Based on the provided Context, please follow these steps:
[numbered steps]

Context: ...

This template is broken down into the following components.

Each agent in our framework is defined by four key components: the Role, which designates the agent's specific function (such as Auditor, Architect, or Code Generator); the Task, which outlines the agent's particular objectives; the Instruction, which provides detailed step-by-step guidance using chain-of-thought reasoning; and the Context, which encompasses all necessary information including input code, audit reports, architectural plans, RAG datastore examples, and inter-agent conversation history.

465 Table 3 shows how this template is adapted for each agent.

466 **Table 3: Agent Prompts for Smart Contract Repair.**

Role	Task	Instruction	Context
Auditor	Identify vulnerabilities and unsafe patterns in Solidity/Move code.	Analyze the code for security vulnerabilities and generate a detailed report.	Input smart contract code (Solidity/-Move).
Architect	Create a high-level plan to address vulnerabilities identified by the Auditor.	Review the Auditor's report and develop a plan outlining necessary modifications.	Auditor's report.
Code Generator	Generate Repaired Solidity/Move code based on the Architect's plan and RAG examples.	Consult the Architect's plan, retrieve examples from the RAG datastore, and generate repaired code.	Architect's plan, Solidity/Move code examples from RAG.
Refiner	Iteratively refine the generated code to improve quality and efficiency.	Review the generated code, identify areas for improvement, and refine accordingly.	Generated code, previous iteration code (if any).
Validator	Perform a final security check on the repaired code.	Analyze the repaired code for vulnerabilities, verify issue resolution, and ensure no new vulnerabilities.	Repaired smart contract code.

4.2 Hardware and Model Fine-tuning

The development and deployment of Smartify leveraged a heterogeneous computing environment, utilizing high-performance GPUs for computationally intensive tasks and a more resource-efficient setup for inference.

4.2.1 Fine-tuning Setup

- Hardware:** Fine-tuning leveraged a cluster of **four NVIDIA A100 GPUs** for computationally demanding pattern learning in Solidity and Move code.
- Model:** Based on the **Gemma 9B model**, selected for strong code-related task performance and fine-tuning adaptability, particularly in instruction following. Fine-tuned on a dataset of Solidity and Move code, vulnerability examples, best practices, and documentation, augmented with outputs from earlier pipeline stages to enhance safety issue detection.
- Training Recipe:** Supervised learning paradigm. Trained to predict correct outputs (e.g., vulnerability reports, safe code patterns) from inputs (e.g., Solidity/Move code, vulnerability descriptions).
 - Data Pre-processing:** Tokenization, normalization, and input-output pair creation ensured data consistency and quality.
 - Hyperparameter Optimization:** Learning rate (1e-5), batch size (8, due to memory constraints), and training epochs (5, as validation loss plateaued) optimized via grid search and manual tuning.
 - Regularization:** Dropout and weight decay used to prevent overfitting and improve generalization.
 - Evaluation Metrics:** Accuracy, precision, recall, and F1-score on a held-out validation set monitored model performance.

4.2.2 Inference Setup

- Hardware:** Inference was performed on a single **NVIDIA RTX 4090 GPU**, balancing performance and cost-effectiveness for real-time code repair.

- Models:**

- Code Generator and Refiner:** These agents utilize a fine-tuned **CodeGemma** model, initially pre-trained on a limited Move corpus and further instruction-tuned to follow Architect-generated "recipe" patterns. Fine-tuning on Architect outputs ensured it understood these instructions, and pre-training on a limited Move corpus ensured basic syntax understanding.
- Comparison Model:** A stock **Llama 3.1** model was used in some experiments for comparative analysis, helping assess the gains from fine-tuning and instruction tuning.

4.2.3 Key Considerations

- A balance between performance requirements, resource availability, and cost considerations drove the choice of hardware and models.
- The fine-tuning process for the *Auditor* was particularly resource-intensive due to the complexity of the task and the size of the model.
- The use of a smaller, more efficient GPU for inference makes the system more accessible for practical deployment.
- The comparison with a stock Llama 3 model provides valuable insights into the effectiveness of our fine-tuning and instruction-tuning strategies.

This heterogeneous setup, combining high-performance GPUs for training and a more efficient GPU for inference, allows Smartify to effectively address the computational demands of both model development and deployment. The detailed description of the fine-tuning process provides transparency and allows for replication of our results.

5 Experimental Results and Discussion

We run our experiments as defined in Section 3.3. We report the results as well as the empirical performance of our models. Through that, we will try to answer our Research Questions individually in this section.

Along with Smartify, we have run the benchmark for the following models.

563 **Table 4: Comparison of Code and Non-Code Models.**

Model Name	Parameters	Quantization	Code Model
Granite-Code	8B	FP16	Yes
CodeGemma	7B	FP16	Yes
DeepSeek-Coder-v2	N/A	N/A	Yes
StarCoder2	15B	FP16	Yes
CodeGeex4	13B	N/A	Yes
CodeStral	7B	FP16	Yes
DeepSeek-Coder	33B	N/A	Yes
CodeLlama [32]	13B	N/A	Yes
CodeQwen	7B	Q8_0	Yes
Qwen2.5-coder	2.5B	N/A	Yes
Gemma2	N/A	N/A	Yes
Gemma2:27b	27B	FP16	Yes
Llama3.2	3.2B	FP16	No
OpenCoder	8B	FP16	Yes
Llama3.3	3.3B	FP16	No

The models were chosen according to the top 8 models at Hugging Face Big Code Leaderboard [1] at the time of this work, and

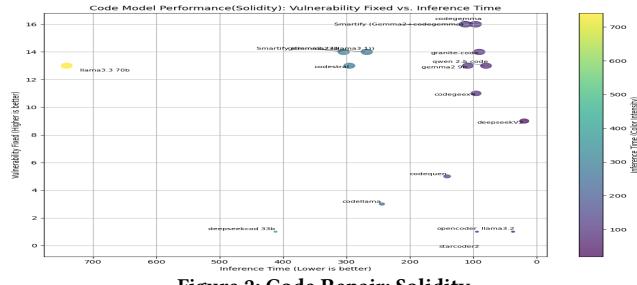


Figure 2: Code Repair: Solidity.

also adding general-purpose models, which are supposed to be better at reasoning.

5.1 Solidity

This section presents the evaluation results of various code generation models on repairing vulnerabilities in Solidity smart contracts, specifically focusing on the “Not So Smart Contracts” dataset from the Trail of Bits GitHub repository. This dataset is a collection of intentionally vulnerable Solidity contracts designed to test the ability of automated tools to detect and repair common security flaws. It contains diverse vulnerabilities, including re-entrancy, integer overflow/underflow, access control issues, and timestamp dependence, among others. The dataset has been publicly available for a significant period, raising the possibility that some or all of its contents might be present in the pre-training data of the evaluated models. We analyze the performance of these models based on two key metrics: the number of vulnerabilities fixed and the average inference time, as summarized in Table 5 and Figure 2. We also introduce our framework, Smartify, and demonstrate its effectiveness in enhancing model performance.

Table 5: Performance of Code Generation Models on Vulnerability Repair.

Model Name	Vuln. Fixed	Avg. Time (s)
CodeGeex-4	11	95.50
CodeGemma	16	96.50
CodeLlama	3	243.93
CodeQwen	5	141.05
CodeStral	13	295.23
DeepSeekCoder-33b	1	411.75
DeepSeek-V2	9	19.42
Gemma2-9b	13	108.30
Gemma2-27b	14	304.27
Granite-Code	14	90.37
Llama3.2	1	37.09
Llama3.3-70b	13	741.10
OpenCoder*	1*	94*
Qwen-2.5-Code	13	79.72
StarCoder2	0*	89.10
Smartify (Gemma2+CodeGemma)	16	112.30
Smartify (Gemma2+Llama3.1)	14	267.80

The results reveal significant performance disparities among the evaluated models. Among the pre-trained models for Solidity **CodeGemma** surprisingly emerges as a top performer, successfully fixing 16 vulnerabilities with a relatively low average inference time of 96.5 seconds. This suggests that **CodeGemma** possesses a strong ability to understand and rectify code vulnerabilities while maintaining reasonable efficiency. However since most of these Solidity smart contracts were part of open Githubs repositories, there can be a strong possibility fo these already being part of the pertaining data. Our proposed framework, **Smartify (Gemma2+CodeGemma)**, achieves comparable performance, also fixing 16 vulnerabilities, albeit with a slightly higher average inference time of 112.3 seconds.

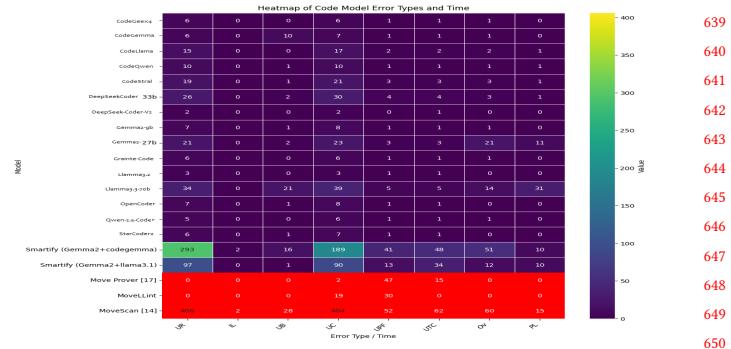


Figure 3: Move Code Repair.

This increased time is likely due to its iterative multi-agent process, which enables Smartify to leverage the complementary strengths of **Gemma2** and **CodeGemma**, resulting in robust and reliable fixes.

While Smartify does not immediately show any benefits over **CodeGemma** here, we can notice that the same Smartify framework when applied to **Llama3.1** without any fine-tuning (unlike the Smartify with **CodeGemma**) still gives considerable performance boost over vanilla.

Conversely, models like **CodeLlama**, **CodeQwen**, **DeepSeekCoder-33b**, and **Llama3.2** show limited effectiveness, fixing only a small number of vulnerabilities. The poor performance of these models could be attributed to several factors, such as insufficient exposure to Solidity code during pre-training or fine-tuning or architectures ill-suited for vulnerability repair, which requires a deep understanding of code syntax and security principles. The exceptionally poor performance of models like **starcoder2** (marked with an asterisk *), along with incomplete data for **opencoder**, suggests potential issues with their training data or a fundamental mismatch between their capabilities and the task's demands. These models might have been trained on an older version of Solidity or different smart contract security practices than those in the Not-So-Smart-Contracts dataset. Moreover, they might prioritize other aspects of code generation, such as code completion, over security-specific tasks like vulnerability repair.

Table 6: Move Vulnerability Repair (Time in seconds).

Model	UR	IL	UB	UC	UPF	UTC	Ov	PL	Time
CodeGeex4	6	0	0	6	1	1	1	0	96
CodeGemma	6	0	10	7	1	1	0	0	97
CodeLlama	15	0	1	17	2	2	1	244	682
CodeQwen	10	0	1	10	1	1	1	141	683
CodeStral	19	0	1	21	3	3	2	1	295
DeepSeekCoder-33b	26	0	2	30	4	4	3	1	412
DeepSeekV2	2	0	0	2	0	1	0	0	19
Gemma2-9b	7	0	1	8	1	1	0	10	108
Gemma2-27b	21	0	2	23	3	3	21	11	304
Granite-Code	6	0	0	6	1	1	1	0	90
Llama3.2	3	0	0	3	1	1	0	0	37
Llama3.3-70b	34	0	21	39	5	5	14	31	741
OpenCoder	7	0	1	8	1	1	0	0	94*
Qwen-2.5-Code	5	0	0	6	1	1	1	0	80
StarCoder2	6	0	1	7	1	1	0	0	89
Smartify (Gemma2+CodeGemma)	293	2	16	189	41	48	51	10	112
Smartify (Gemma2+Llama3.1)	97	0	1	90	13	34	12	10	268
Move Prover [11]	-	2	-	-	-	-	47	15	-
MoveLint	-	-	-	19	30	0	0	-	-
MoveScan [33]	406	2	28	404	52	62	60	15	-

Abbreviations: UR: Unchecked Return; IL: Infinite Loop; UB: Unnecessary Boolean; UC: Unused Constant; UPF: Unused Private Function; UTC: Unnecessary Type Conversion; Ov: Overflow; PL: Precision Loss.

697 The public availability of the "Not So Smart Contracts" dataset
 698 raises the question of data contamination. Many evaluated models,
 699 especially those trained on large, public code corpora, might have
 700 encountered this dataset during pre-training, potentially inflating
 701 their performance. However, since **CodeGemma** and **Smartify**
 702 (**Gemma2+CodeGemma**) were specifically fine-tuned for this
 703 task, the issue of data contamination is likely less significant.

704 5.2 Move Code Repair

705 This section analyzes the efficacy of various models in repairing
 706 vulnerabilities within Move smart contracts, as detailed in Table 6.
 707 The evaluation encompasses eight distinct vulnerability categories:
 708 *Unchecked Return (UR)*, *Infinite Loop (IL)*, *Unnecessary Boolean (UB)*,
 709 *Unused Constant (UC)*, *Unused Private Function (UPF)*, *Unnecessary*
 710 *Type Conversion (UTC)*, *Overflow (Ov)*, and *Precision Loss (PL)* fol-
 711 lowing the works of Song et al [33]. The metrics presented in the
 712 table represent the number of successfully repaired instances for
 713 each vulnerability type, with higher values indicating superior per-
 714 formance. The inference time, measured in seconds, is also provided
 715 for each model.

716 The results demonstrate a significant variance in performance
 717 across the evaluated models. Notably, the larger language models,
 718 such as **Deepseekcoder 33b** and **Llama3.3 70b**, exhibit a relatively
 719 higher number of successful repairs across multiple categories, al-
 720 albeit with a corresponding increase in inference time. Conversely,
 721 smaller models like **DeepseekV2** and **Llama3.2** demonstrate lim-
 722 ited repair capabilities. The specialized tools for Move code, namely
 723 **Move Prover**, **MoveLint**, and **MoveScan**, were employed as a
 724 benchmark for comparison. It is crucial to note that these tools are
 725 designed for vulnerability **detection** rather than repair. **MoveScan**,
 726 in particular, identified a substantial number of instances across
 727 all categories, highlighting its effectiveness as a static analysis tool.
Move Prover demonstrated proficiency in detecting Overflow and
 728 Precision Loss vulnerabilities, while **MoveLint** focused on Unused
 729 Private Functions and Unnecessary Type Conversions.

730 The Smartify models, which take advantage of a combination
 731 of **Gemma2** with either **CodeGemma** or **Llama3.1**, present an
 732 interesting case. Smartify(**Gemma2+CodeGemma**) and Smartify(**Gemma2+Llama3.1**) outperform several individual models in
 733 multiple categories. This is likely because the specialized models
 734 are fine-tuned on the Move-specific dataset. For instance, Smartify
 735 (**Gemma2+CodeGemma**) achieves the highest number of repairs
 736 for the Unchecked Return, Infinite Loop, Unused Boolean, Unused
 737 Constant, Unused Private Function, Unnecessary Type Conversion,
 738 and Overflow categories, showcasing a substantial improvement
 739 over individual models in these areas. However, it is worth men-
 740 tioning that they also have limitations compared to individual models
 741 for certain categories like Precision Loss.

742 RQ1 & RQ2 - Code Understanding and Vuln. Detection

743 Yes. Our empirical analysis with Smartify, especially with using a fine-tuned
 744 **Code-Gemma** and also using vanilla pre-trained **Llama3.1**, has shown us the
 745 effectiveness of the framework's ability to understand code and capture bad
 746 practices leading to vulnerability. Especially for a low-resource code like Move,
 747 without significant fine-tuning (in the case of **Llama3.1**), Smartify outperforms
 748 the larger and computationally intensive models such as Llama 3.3 70b.

749 Notably, **Smartify (Gemma2+CodeGemma)**, combining fine-
 750 tuned **Gemma2** with **CodeGemma**, achieves performance on par
 751 with the best individual model, **CodeGemma**, which is expected
 752 due to one of the models being fine-tuned. This highlights the ad-
 753 vantages of strategically combining specialized models, answering
 754 our next research question.

755 RQ3 & RQ4 - Code Repair

756 Both for Solidity and Move, we were able to compare the efficacy of our frame-
 757 work with prior works. We can see Smartify outperforms all of the existing code
 758 models, even very specialized code models trained on Move (OpenCoder [17]) in
 759 generating repair codes for detected vulnerabilities.

760 Furthermore, Smartify's efficacy extends even when integrat-
 761 ing a non-finetuned model like **Llama 3.1**. Smartify significantly
 762 outperforms **Llama 3.2** by fixing 14 vulnerabilities compared to
 763 Llama 3.2's single fix, making its performance comparable with the
 764 much more extensive and computationally intensive **Llama 3.3**
765 70b. This demonstrates that Smartify's architecture can enhance
 766 even general-purpose language models for code repair, balancing
 767 speed, and accuracy.

768 RQ5 - Generalization

769 Our implementation of Smartify with both fine-tuned **Code-Gemma** and
 770 **Llama3.1** as the second agent allowed us to run our experiments on both sets of
 771 LLMs. The results show that Smartify can significantly boost performance even
 772 on non-finetuned models compared to a single model.

773 Comparative analysis reveals trade-offs between model scale
 774 and performance in automated code repair. Larger models, such as
 775 **DeepSeekCoder 33b** and **Llama 3.3 70b**, exhibit broader repair
 776 capabilities but incur higher computational costs and inference
 777 times. Conversely, the **Gemma2 27b** model demonstrates notable
 778 proficiency in addressing Overflow vulnerabilities, albeit with limi-
 779 tations in handling Unnecessary Boolean and Unused Constant com-
 780 pared to **Llama 3.3 70b**. While **Llama 3.3 70b** outperforms Smartify
 781 in overall repair capability, its significantly slower inference
 782 speed poses a challenge for practical deployment. Therefore, for real-
 783 world, on-device applications, **Smartify (Gemma2+CodeGemma)**
 784 presents a compelling solution with its balance of substantial accu-
 785 racy and rapid inference.

786 **Insight:** Specialized code models like **StarCoder** [25], **OpenCoder** [17]
 787 and **DeepSeekCoder** [14] doesn't necessarily work well even if it's a
 788 coding specific task. While code models like **CodeGemma** [34] and
 789 **CodeLlama** [32] are much better at understanding instructions and
 790 working on code. This helped Smartify for its understanding and fine-
 791 tuning for code repairability.

792 Specialized static analysis tools for Move, including **Move Prover**,
 793 **MoveLint**, and **MoveScan**, work as baselines of detecting Move vul-
 794 nerabilities with which we compare our Smartify and other LLMs.
 795 These findings underscore the need for targeted model improve-
 796 ments. The Smartify framework directly addresses these deficien-
 797 cies, offering enhanced vulnerability repair effectiveness.

798 This research also opens up future research directions on using
 799 this framework for context-aware test case generation.

813 6 Ablation Study

814 To rigorously validate our architectural design and isolate the
 815 contribution of each key component, a comprehensive ablation study
 816 was conducted. This study systematically deconstructs the Smartify
 817 framework to quantify the impact of its core mechanisms—agent
 818 specialization, iterative refinement, and retrieval-augmented gen-
 819 eration—on overall repair performance [16, 19].
 820

821 6.1 Ablation Configurations

822 Four distinct configurations of the system were evaluated against
 823 the Solidity and Move datasets:
 824

- 825 • **Full Smartify:** The complete five-agent system, including the
 826 RAG module for the Code Generator and the Refiner-Validator
 827 iterative feedback loop. This represents our proposed approach.
 828
- 829 • **Smartify (No Refinement):** A version of the framework where
 830 the Refiner and Validator agents are disabled. The output is taken
 831 directly from the first pass of the Code Generator. This config-
 832 uration measures the impact of the iterative self-improvement
 833 loop on patch quality.
 834
- 835 • **Smartify (No RAG):** In this setup, the Code Generator oper-
 836 ates without the contextual, few-shot examples provided by the
 837 RAG system. It relies solely on its fine-tuned knowledge and the
 838 Architect’s plan. This configuration is designed to measure the
 839 importance of providing language-specific, in-context examples,
 840 especially for the low-resource Move language.
 841
- 842 • **Single-Agent Baseline (CoT-style):** This configuration replaces
 843 the multi-agent system with a single, powerful agent (Gemma2
 844 9B). The agent is given a complex, chained prompt that instructs
 845 it to sequentially perform the tasks of auditing, planning, and
 846 generating the repair. This mimics the linear reasoning work-
 847 flow of CoT-based approaches like ContractTinker and directly
 848 tests the value of our role-based specialization and delegation
 849 architecture.

850 6.2 Analysis

851 The performance of each configuration was measured using the
 852 Pass@1 and Exploit Mitigation Rate metrics. The results, presented
 853 in Table 7, provide clear empirical evidence supporting our archi-
 854 tectural choices.
 855

856 **Table 7: Ablation Study of Smartify Components on Solidity and**
 857 **Move Datasets**

858 Configuration	Solidity		Move	
	Pass@1 (%)	Exploit Mit. (%)	Pass@1 (%)	Exploit Mit. (%)
Full Smartify	26.7	25.0	48.9	45.7
Smartify (No Refinement)	25.0	20.0	44.6	39.1
Smartify (No RAG)	21.7	18.3	28.3	23.9
Single-Agent Baseline	18.3	15.0	21.7	17.4

863 The results demonstrate a clear and consistent degradation in
 864 performance as key components are removed, confirming the pos-
 865 tive contribution of each element of the Smartify architecture.
 866

- 867 • **Impact of Iterative Refinement:** Removing the refinement
 868 loop (**No Refinement**) causes a noticeable drop in both metrics,
 869 particularly the Exploit Mitigation Rate (from 25.0% to 20.0% for
 870 Solidity). This indicates that while the initial code generation is

often correct, the refinement process is crucial for hardening the
 871 patch against exploits and catching subtle regressions that the
 872 first pass might miss.
 873

- 874 • **Impact of RAG:** The removal of the RAG module (**No RAG**)
 875 has a significant negative impact across the board, but its effect
 876 is most pronounced for the Move language. The Pass@1 score
 877 for Move plummets from 48.9% to 28.3%, a relative decrease of
 878 over 42%. This strongly supports our hypothesis that providing
 879 in-context, language-specific examples is critical for achieving
 880 high performance in low-resource languages where the model’s
 881 pre-trained knowledge is limited.
 882

- 883 • **Impact of Multi-Agent Architecture:** The **Single-Agent Base-
 884 line**, designed to mimic a CoT-style approach, performs the worst
 885 of all configurations. Its performance is substantially lower than
 886 the Full Smartify framework, particularly on the more complex
 887 task of exploit mitigation. This finding provides strong evidence
 888 that our role-based, delegative architecture is superior to a mono-
 889 lithic, linear reasoning process. By specializing agents for distinct
 890 tasks (analysis vs. planning vs. generation), the system achieves a
 891 more robust and effective decomposition of the complex program
 892 repair problem.
 893

894 The ablation study empirically validates the design of Smartify.
 895 The multi-agent architecture provides a superior structure for com-
 896 plex problem-solving, the refinement loop is essential for ensuring
 897 patch quality and security, and the RAG mechanism is a critical
 898 component for adapting the framework to new or low-resource
 899 programming languages.
 900

7 Conclusion

901 This work addresses the pressing need for enhanced security in the
 902 burgeoning blockchain ecosystem. We investigate the application
 903 of Large Language Models (LLMs) to smart contract vulnerability
 904 detection and repair, focusing on Solidity and Move. We intro-
 905 duce **Smartify**, a novel multi-agent framework that significantly
 906 improves LLM performance in this critical domain. The contribu-
 907 tions of this work are: (1) **Smartify**, a novel multi-agent framework
 908 that enhances LLM-based smart contract vulnerability detection
 909 and repair; (2) a method for encoding language-specific knowl-
 910 edge, valuable for low-resource languages like Move; (3) a scalable,
 911 adaptable approach applicable to other programming languages
 912 and LLMs; (4) a demonstration of Smartify’s efficacy on general-
 913 ized pre-trained LLMs; and (5) a detailed analysis of the challenges
 914 inherent in automated code repair.
 915

916 **Smartify** represents a significant advancement in automating
 917 smart contract security, a crucial concern in the expanding blockchain
 918 landscape. Future work will refine the framework, expand its lan-
 919 guage coverage, particularly within the blockchain domain, and
 920 integrate it into real-world blockchain development workflows.
 921 This research lays the foundation for AI-powered tools that can
 922 bolster the security and reliability of decentralized applications,
 923 fostering a more robust and trustworthy blockchain ecosystem.
 924

925 References

- [1] [n. d.]. Big Code Models Leaderboard - a Hugging Face Space by bigcode – huggingface.co. <https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>.
 926

- [Accessed 10-01-2025].

[2] [n.d.]. not-so-smart-contracts: Examples of Solidity security issues. <https://github.com/cryptic/not-so-smart-contracts>. [Accessed 09-01-2025].

[3] 2024. Starcoin Whitepaper. https://starcoin.org/_astro/whitepaper.CMIZt6_x.pdf

[4] 2025. Decoding Poly Network \$34 Billion Hack. <https://www.quillaudits.com/blog/hack-analysis/poly-network-hack>

[5] Sam Blackshear, Evan Cheng, David L Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Dario Russi Rain, Stephane Sezer, et al. 2019. Move: A language with programmable resources. *Libra Assoc* (2019), 1.

[6] Sam Blackshear, Andrew Chursin, George Danezis, Anastasios Kichidis, Lefteris Kokoris-Kogias, Xun Li, Mark Logan, Ashok Menon, Todd Nowacki, Alberto Sonnino, et al. 2024. Sui lutris: A blockchain combining broadcast and consensus. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*. 2606–2620.

[7] Sam Blackshear, Todd Nowacki, David Dill, Avery Ching, Chih-Cheng Liang, and Tim Zakiyan. 2022. The Move Borrow Checker. arXiv:2205.05181 <https://arxiv.org/abs/2205.05181> arXiv preprint.

[8] Sofia Bobadilla, Monica Jin, and Martin Monperrus. 2025. Do Automated Fixes Truly Mitigate Smart Contract Exploits? arXiv:2501.04600 <https://arxiv.org/abs/2501.04600> arXiv e-prints.

[9] Chris Dannen and Chris Dannen. 2017. Solidity programming. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners* (2017), 69–88.

[10] Aptos Dev. [n.d.]. The aptos blockchain: Safe, scalable, and upgradeable web3 infrastructure.

[11] David Dill, Wolfgang Grieskamp, Junkil Park, Shaz Qadeer, Meng Xu, and Emma Zhong. 2022. Fast and reliable formal verification of smart contracts with the move prover. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 183–200.

[12] Josselin Feist, Gustavo Grieco, and Alex Groce. 2019. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*. IEEE, 8–15.

[13] Gustavo Grieco, Santiago Palladino, Martin Ortizaga, Federico Bond, and Valentin Wustholz. 2020. Echidna: Effective, Usable, and Fast Fuzzing for Smart Contracts. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20)*. ACM, 481–493. <https://doi.org/10.1145/3395363.3397384>

[14] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Yu Wu, YK Li, et al. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming—The Rise of Code Intelligence. arXiv preprint arXiv:2401.14196 (2024).

[15] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. 2018. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 204–217.

[16] Siming Huang, Tianhao Cheng, J. K. Liu, Jiaran Hao, Liuyihan Song, Yang Xu, J. Yang, Jiabing Liu, Chenchen Zhang, Linzheng Chai, Ruifeng Yuan, Zhaoxiang Zhang, Jie Fu, Qian Liu, Ge Zhang, Zili Wang, Yuan Qi, Yinghui Xu, and Wei Chu. 2025. OpenCoder: The Open Cookbook for Top-Tier Code Large Language Models. arXiv:2411.04905 [cs.CL] <https://arxiv.org/abs/2411.04905>

[17] Siming Huang, Qingyue Zhang, Yilun Jin, Xiaozhu Meng, Yiyang Wang, Haoyu Wang, Ding Li, and Shiqing Ma. 2024. Opencoder: The Open Cookbook for Top-Tier Code Large Language Models. arXiv:2411.04905 <https://arxiv.org/abs/2411.04905> arXiv preprint.

[18] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. <https://doi.org/10.1145/3238147.3238177>

[19] Bo Jiang, Ye Liu, and W. K. Chan. 2018. ContractFuzzer: fuzzing smart contracts for vulnerability detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (Montpellier, France) (ASE '18)*. Association for Computing Machinery, New York, NY, USA, 259–269. <https://doi.org/10.1145/3238147.3238177>

[20] Harshit Joshi, José Cambrónero Sanchez, Sumit Gulwani, Vu Le, Gust Verbruggen, and Ivan Radíček. 2023. Repair is nearly generation: Multilingual program repair with llms. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 37. 5131–5140.

[21] Mudabbir Kaleem, Keshav Kasichainula, Rabimba Karanjai, Lei Xu, Zhimin Gao, Lin Chen, and Weidong Shi. 2021. An event driven framework for smart contract execution. In *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*. 78–89.

[22] Rabimba Karanjai, Lei Xu, Zhimin Gao, Lin Chen, Mudabbir Kaleem, and Weidong Shi. 2021. On conditional cryptocurrency with privacy. In *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. IEEE, 1–3.

[23] Eric Keilty, Keerthi Nelaturu, Bowen Wu, and Andreas Veneris. 2022. A model-checking framework for the verification of move smart contracts. In *2022 IEEE 13th International Conference on Software Engineering and Service Science (ICSESS)*.

[24] Jinggang Li, Gehao Lu, Yulian Gao, and Feng Gao. 2023. A Smart Contract Vulnerability Detection Method Based on Multimodal Feature Fusion and Deep Learning. *Mathematics* 11, 23 (2023), 4823.

[25] Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtiar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocevsk, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indranil Paul, Zhang Huang, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muenmighoff, Xiangru Tang, Muh-tasham Oblikulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebazé, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados, Mostofa Patwary, Nima Tajbakhsh, Yacine Jernite, Carlos Muñoz Ferrandis, Lingming Zhang, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. 2024. StarCoder 2 and The Stack v2: The Next Generation. arXiv:2402.19173 [cs.SE]

[26] Mark Mossberg, Felipe Manzano, Eric Hennenfent, Alex Groce, Gustavo Grieco, Josselin Feist, Trent Brunson, and Artem Dinaburg. 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1186–1189.

[27] Bernhard Mueller. [n.d.]. File 1 of. ([n. d.]).

[28] Keshab Nath, Sourish Dhar, and Subhash Basishtha. 2014. Web 1.0 to Web 3.0—Evolution of the Web and its various challenges. In *2014 International Conference on Reliability Optimization and Information Technology (ICROIT)*. IEEE, 86–89.

[29] Junkil Park, Teng Zhang, Wolfgang Grieskamp, Meng Xu, Gerardo Di Giacomo, Kundu Chen, Yi Lu, and Robert Chen. 2024. Securing Aptos framework with formal verification. In *5th International Workshop on Formal Methods for Blockchains (FMBC 2024)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[30] Nathaniel Popper. 2016. A hacking of more than \$50 million dashes hopes in the world of virtual currency. *The New York Times* 17 (2016).

[31] Partha Pratim Ray. 2023. Web3: A comprehensive review on background, technologies, applications, zero-trust architectures, challenges and future directions. *Internet of Things and Cyber-Physical Systems* 3 (2023), 213–248.

[32] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaqing 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 (2023).

[33] Shuwei Song, Jiachi Chen, Ting Chen, Xiapu Luo, Teng Li, Wenwu Yang, Leping Wang, Weijie Zhang, Feng Luo, Zheyuan He, et al. 2024. Empirical Study of Move Smart Contract Security: Introducing MoveScan for Enhanced Analysis. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1682–1694.

[34] CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqu Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyu Shen, Joe Kelley, et al. 2024. Codgemma: Open code models based on gemma. arXiv preprint arXiv:2406.11409 (2024).

[35] Anna Vacca, Andrea Di Sorbo, Corrado A Visaggio, and Gerardo Canfora. 2021. A systematic literature review of blockchain and smart contract development: Techniques, tools, and open challenges. *Journal of Systems and Software* 174 (2021), 110891.

[36] Che Wang, Jiashuo Zhang, Jianbo Gao, Libin Xia, Zhi Guan, and Zhong Chen. 2024. Contracttinker: Llm-empowered vulnerability repair for real-world smart contracts. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 2350–2353.

[37] Wei Wang, Jingjing Song, Guangquan Xu, Yidong Li, Hao Wang, and Chunhua Su. 2020. Contractward: Automated vulnerability detection models for ethereum smart contracts. *IEEE Transactions on Network Science and Engineering* 8, 2 (2020), 1133–1144.

[38] Christoph Wronka. 2023. Financial crime in the decentralized finance ecosystem: new challenges for compliance. *Journal of Financial Crime* 30, 1 (2023), 97–113.

[39] Zibin Zheng, Shaona Xie, Hong-Ning Dai, Xiangping Chen, and Huaimin Wang. 2018. Blockchain challenges and opportunities: A survey. *International journal of web and grid services* 14, 4 (2018), 352–375.

[40] Jingyi Emma Zhong, Kevin Cheang, Shaz Qadeer, Wolfgang Grieskamp, Sam Blackshear, Junkil Park, Yoni Zohar, Clark Barrett, and David L Dill. 2020. The move prover. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I* 32. Springer, 137–150.