SCALM: Detecting Bad Practices in Smart Contracts Through LLMs

Zongwei Li, Xiaoqi Li*, Wenkai Li, Xin Wang

School of Cyberspace Security, Hainan University, Haikou, 570228, China
lizw1017@gmail.com, csxqli@ieee.org, liwenkai871@gmail.com, wxin98767@gmail.com

Abstract

As the Ethereum platform continues to mature and gain widespread usage, it is crucial to maintain high standards of smart contract writing practices. While bad practices in smart contracts may not directly lead to security issues, they do elevate the risk of encountering problems. Therefore, to understand and avoid these bad practices, this paper introduces the first systematic study of bad practices in smart contracts, delving into over 35 specific issues. Specifically, we propose a large language models (LLMs)-based framework, SCALM. It combines Step-Back Prompting and Retrieval-Augmented Generation (RAG) to identify and address various bad practices effectively. Our extensive experiments using multiple LLMs and datasets have shown that SCALM outperforms existing tools in detecting bad practices in smart contracts.

1 Introduction

With the widespread use of blockchain technology, smart contracts have become an important part of the blockchain ecosystem (Sharma, Jindal, and Borah 2023). Smart contracts are computer programs that automatically execute contract terms, controlling assets and operations on the chain. However, due to their public and immutable code, smart contracts have become a significant target for attackers. A total of 464 security incidents occurred in 2023, resulting in losses of up to $2.486 billion (Zone 2024). The most significant attack occurred on September 23rd when Mixin Network's cloud service provider database was attacked, involving approximately $200 million.

Bad practices refer to poor coding habits or design decisions in the development of smart contracts. Although bad practices may not result in immediate security threats, they could potentially lead to future issues such as performance problems, increased security vulnerabilities, and unpredictable code behavior. Additionally, bad practices have hidden economic dangers due to the disruption of regular smart contract activities.

Currently, the security audit of smart contracts mainly relies on manual code review and automated tools. However, these methods have their limitations (Chaliasos et al. 2024). Manual code review is inefficient and prone to overlook subtle security vulnerabilities. Existing automated tools primarily rely on pattern matching, which cannot accurately detect complex security issues. Moreover, the types of vulnerabilities that these tools can detect are usually relatively limited. To achieve a comprehensive audit, multiple tools may be required, each covering different aspects of security.

To solve these problems, we introduce SCALM (**S**mart **C**ontract **A**udit **L**anguage **M**odel), a new security auditing framework for smart contracts. It mainly consists of two parts: first, it performs static analysis on large datasets to identify and extract code blocks containing potential bad practices. These are then vectorized and stored in a vector database as a searchable knowledge base. Second, SCALM utilizes RAG and Step-Back Prompting to abstract high-level concepts and principles from the code, enabling the detection of bad practices. The framework ultimately generates detailed audit reports that highlight security issues, assess associated risks, and provide actionable remediation recommendations.

Our contributions are as follows:

- To the best of our knowledge, we provide the first systematic study of bad practices in smart contracts and conduct an in-depth discussion and analysis on 35 different types of SWC covered.
- We introduce SCALM, a novel framework based on LLMs for smart contract security audits. It mainly consists of two parts: firstly, static code analysis is used to extract code blocks containing bad practices, which are transformed into vectors and stored in a vector database as a knowledge base; secondly, using RAG and Step-Back Prompting to extract high-level concepts and principles from the code to detect bad practices, and ultimately generate detailed audit reports.
- We conduct an experimental evaluation on SCALM, and the results show that the framework performs well and outperforms existing tools in detecting bad practices in smart contracts. At the same time, ablation experiments reveal that the RAG component significantly improves SCALM performance.
- We open source SCALM's codes and experimental data at https://figshare.com/s/5cc3639706e4ecd16724.

*Corresponding author: Xiaoqi Li

Copyright © 2025, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

2 Background

2.1 Large Language Models

LLMs are trained using deep learning techniques to understand and generate human language. They are typically based on the Transformer architecture, such as ChatGPT (Kasneci et al. 2023), BERT (Devlin et al. 2019), GLM (Du et al. 2022), etc. The training process of these models usually involves learning language patterns and structures from large-scale text corpora. These corpora can include a variety of texts such as news articles, books, web pages, and other forms of human linguistic expression. The model can generate coherent and meaningful text by learning from these corpora. In addition, they can handle various natural language processing tasks, including text generation, text classification, sentiment analysis, question-answering systems, etc.

One of the key features of LLMs is their powerful generative ability. These models can generate new, coherent text similar in grammar and semantics to the training data. This makes LLMs useful for various applications, including machine translation, text summarization, sentiment analysis, dialogue systems, and other natural language processing tasks.

Another important feature of LLMs is their ”zero-shot” capability, which allows them to perform various tasks without any task-specific training. For example, the model can choose the most appropriate answer given a question and some answer options. This ability makes LLMs very useful in many practical applications.

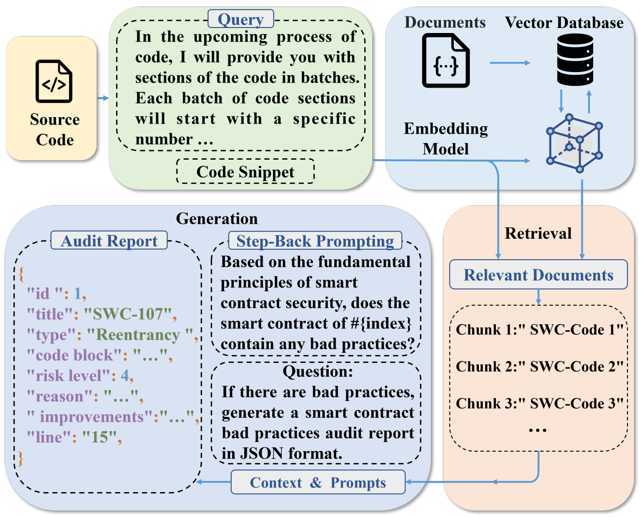
However, LLMs also have some challenges and limitations. For instance, they may generate inaccurate or misleading information and reflect biases in the training data. Moreover, due to their need for substantial computational resources for training and operation, they might face certain challenges in practical application.

2.2 Smart Contract Weakness Classification

Smart contracts are self-executing protocols that run on the blockchain and allow trusted transactions without third-party intervention. However, since their code is publicly available and cannot be changed once deployed, the security of smart contracts has become an important issue. To address this issue, EIP-1470 (Wagner 2018) proposes the Smart Contract Weakness Classification (SWC), a tool designed to help developers identify and prevent smart contract weaknesses.

SWC concerns weaknesses that can be identified within a smart contract's Solidity code. It is designed to reference the structure and terminology of the Common Weakness Enumeration (CWE) but adds several weakness classifications specific to smart contracts. These classifications include but are not limited to, reentry attacks, arithmetic overflow, Assert Violation, etc.

All work on SWC has been incorporated into the EEA EthTrust Security Level Specification, a specification proposed by the Enterprise Ethereum Alliance (EEA) to provide a reliable methodology for assessing the security of smart contracts. This specification defines a series of security levels to measure the security and trustworthiness of smart contracts.



Source Code
Query
In the upcoming process of code, I will provide you with sections of the code in batches. Each batch of code sections will start with a specific number ...
Code Snippet
Documents
Vector Database
Embedding Model
Generation
Audit Report
{
"id ": 1,
"title": "SWC-107",
"type": "Reentrancy ",
"code block": "...",
"risk level": 4,
"reason": "...",
" improvements":"...",
"line": "15",
}
Step-Back Prompting
Based on the fundamental principles of smart contract security, does the smart contract of #{index} contain any bad practices?
Question:
If there are bad practices, generate a smart contract bad practices audit report in JSON format.
Context & Prompts
Retrieval
Relevant Documents
Chunk 1:" SWC-Code 1"
Chunk 2:" SWC-Code 2"
Chunk 3:" SWC-Code 3"
...

Figure 1: Overall Architecture of SCALM.

3 Method

The overview of SCALM's architecture is shown in Fig. 1, which mainly consists of two modules: (1) Extracts defective code blocks via static analysis, converts them into vectors, and stores them in a vector database to create a queryable library of code information. (2) Adopts the RAG methodology, which utilizes Step-Back Prompting to abstract high-level concepts and principles from the code and generates a detailed audit report.

The Algorithm 1 outlines the step-by-step procedure for generating an audit report for smart contract code. The algorithm is divided into several stages: initialization, data collection and processing, detection strategy, and report generation.

3.1 Data Collection and Processing

Our data collection comes from the DAppSCAN database (Zheng et al. 2024b), which includes 39,904 smart contracts with 1,618 SWC weaknesses. First, we extract code snippets with bad practices from these contracts. This extraction process involves static code analysis, using tools to identify and extract code snippets with potential security risks. Then, each code block is converted into a vector through our embedding model and stored in the vector database for subsequent fast matching and retrieval operations.

3.2 Vector Database

A vector database is a particular database that can store and query large amounts of vector data (Hambardzumyan et al. 2022). In the vector database, data is stored as vectors, each typically represented by a set of floating-point numbers. These vectors can represent various data types, such as images, audio, text, etc. In bad practice detection tasks, we use an embedding model (i.e., text-embedding-ada-002) to convert code into vectors and then store these vectors in the vector database. This process can be expressed with the following eq. (1):

Algorithm 1: SCALM Algorithm

```
1: **Input:** Smart contract code $C$, DAppSCAN database $D$
2: **Output:** Audit report $R$ in JSON format
3: **Initialization:**
4: Initialize vector database $VDB$
5: Initialize embedding model $f_{\text{Embedding}}$
6: Initialize LLM with RAG and Step-Back Prompting capabilities
7: **Data Collection and Processing:**
8: for each contract $c \in D$ do
9:     for each code snippet $s \in c$ do
10:        if $s$ contains bad practices then
11:            Convert $s$ to vector $\vec{v}_s$ using $f_{\text{Embedding}}$
12:            Store $\vec{v}_s$ in $VDB$
13:        end if
14:    end for
15: end for
16: **Detection Strategy and Report Generation:**
17: Split $C$ into code fragments $F$
18: Initialize empty report $R$
19: for each fragment $f \in F$ do
20:     Convert $f$ to vector $\vec{v}_f$ using $f_{\text{Embedding}}$
21:     Retrieve similar vectors $\{\vec{v}_i\}$ from $VDB$ using cosine similarity
22:     Retrieve corresponding code snippets $\{s_i\}$ for $\{\vec{v}_i\}$
23:     Perform Step-Back Prompting:
24:     Abstract higher-level concepts or principles from $\{s_i\}$
25:     Generate reasoning and evaluation using LLM
26:     Generate audit results for $f$:
27:     *bad practice ID*, *title*, *type*, *bad code block*, *location*, *risk level*, *reason*, *suggestions*, *line*
28:     Append audit results to $R$
29: end for
30: **Return** $R$
```

$$\vec{v} = f_{\text{Embedding}}(\text{Text}) \tag{1}$$

Where $f_{\text{Embedding}}$ is our embedding model, Text is the input text, and $\vec{v}$ is the outputted vector. This vectorized data storage method significantly improves efficiency in handling it. Firstly, storing data as vectors makes it more compact, thus reducing storage space requirements. Secondly, vectorized data facilitates parallel computing, which is crucial when dealing with large-scale datasets. A vital feature of a Vector Database lies in its ability to perform efficient similarity searches, which are notably advantageous when dealing with high-dimensional datasets. This similarity search can be achieved by calculating cosine similarities between two vectors with the eq. (2):

$$\text{similarity}(A, B) = \frac{A \cdot B}{||A||_2 \cdot ||B||_2} \tag{2}$$

where A and B are two vectors, $A \cdot B$ is their dot product, and $||A|||_2$ and $||B|||_2$ are their second-paradigms.

During the query process, the system first converts user queries into vectors and then performs efficient matching and retrieval in the vector database. In the final stage of the query, the system converts matched results back into a format that users can understand and use.

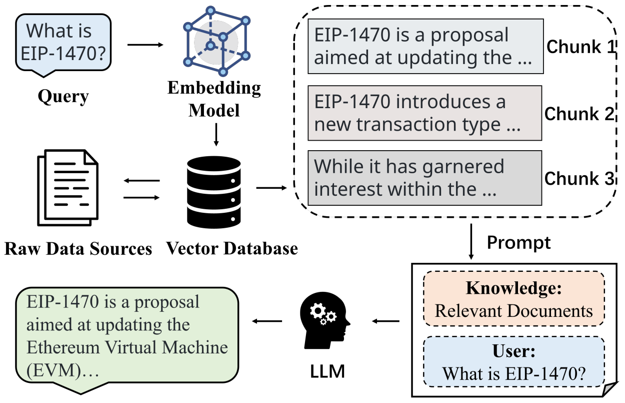

What is EIP-1470?
Query
Embedding Model
EIP-1470 is a proposal aimed at updating the ...
Chunk 1
EIP-1470 introduces a new transaction type ...
Chunk 2
While it has garnered interest within the ...
Chunk 3
Raw Data Sources
Vector Database
Prompt
Knowledge: Relevant Documents
User: What is EIP-1470?
LLM
EIP-1470 is a proposal aimed at updating the Ethereum Virtual Machine (EVM)...

Figure 2: Illustration of the RAG.

3.3 Retrieval-Augmented Generation

LLMs have proven their powerful capabilities in handling complex language understanding and generation tasks, incredibly when fine-tuned for specific downstream tasks. However, these models still face challenges for tasks that require precise and specific knowledge, such as smart contract code auditing.

SCALM adopts the RAG fine-tuning method to improve the quality and accuracy of smart contract code auditing. This method combines pre-trained parametric models with non-parametric memory to enhance the quality and accuracy of smart contract code audits (Gao et al. 2024). The RAG integrates the processes of retrieval and generation into one.

As Fig. 2 illustrates, during the operation of the model, it first retrieves relevant documents or entities from a large-scale knowledge base. Then, it inputs this retrieved information as additional context into the generation model, which generates corresponding outputs based on these inputs. This design allows RAG to utilize external knowledge bases effectively while demonstrating excellent performance when dealing with tasks requiring extensive background knowledge.

3.4 Detection Strategy and Report Generation

Our detection strategy mainly focuses on effectively identifying bad practices in smart contracts. Firstly, the smart contract code to be detected is cut into smaller fragments according to its structure and size, then queried through a vector database to find the most matching known bad practice code blocks.

In the task of detecting bad practices in smart contracts, we adopted a technique called Step-Back Prompting (Zheng et al. 2024a). This technique leverages the capabilities of LLMs to abstract high-level concepts and basic principles from specific code instances. In this way, not only can the model understand the literal meaning of the code, but it can also comprehend underlying logic and potential design patterns through abstract thinking.

Step-Back Prompting consists of two main steps:

- Abstraction: Instead of directly posing questions, we propose a general step-back question about higher-level concepts or principles and retrieve facts related to these high-level concepts or principles. In the task of detecting bad practices in smart contracts, we use abstract prompts that aim to guide LLMs to explore not just the literal meaning of code but deeper structures and intentions. These prompts may include questions like "What are the potential risks with this implementation?"
- Model Reasoning: Based on facts about high-level concepts or principles, LLMs can reason about answers to the original question. We refer to this as abstraction-based reasoning. Reasoning with these abstraction hints attempts to analyze the code from a broader perspective. This includes comparing the strengths and weaknesses of different implementations and how they fit with known best practices or common bad practices.

The LLM evaluates these code snippets for bad practices through Step-Back Prompting and generates a detailed audit report based on the results. The report is output in `JSON` format. It contains the *bad practice ID*, *title*, *type*, specific *bad code block* along with its *location*, *risk level*, *reason* for the problem, and *suggestions* for improvement.

Through this method, we can effectively utilize the powerful capabilities of LLMs for deep security audits on smart contracts, thereby helping developers identify and fix potential security issues.

4 Experiments

4.1 Experiment Settings

All experiments are executed on a server equipped with NVIDIA GeForce GTX 4070Ti GPU, Intel(R) Core(TM) i9-13900KF CPU, and 128G RAM, operating on Ubuntu 22.04 LTS. The software environment includes Python 3.9 and PyTorch 2.0.1.

Dataset. In this paper, we use the **DAppSCAN** dataset (Zheng et al. 2024b) as a knowledge base for detecting bad practices in smart contracts. The dataset contains 39,904 Solidity files with 1,618 SWC weaknesses from 682 real projects. The **Smartbugs** dataset (Durieux et al. 2020) is also used in the experiment, and a total of 1,894 smart contracts with five types of SWC weaknesses are extracted for comparison experiments. These datasets form the basis of our experimental analysis. Table 1 summarizes the smart contract data used.

We conduct initial detection experiments using official SWC samples covering 35 categories of bad practices. It is important to note that these 35 categories of SWC include most bad practices, but they do not represent all possible bad practices that may occur in smart contract development. In the follow-up experiments, we select five categories of SWC for in-depth study, using 94 samples for positive examples of SWC-104 and 200 samples for both positive and negative examples of the remaining SWC categories. In order to comprehensively evaluate the performance of the model on the test set, we select Accuracy (Acc), Recall, and F1 score as the evaluation metrics.

Dataset	# Contracts
DAppSCAN	39,904
Smartbugs	1,894

Table 1: The Collected Dataset for Our Evaluation. # indicates the number of each item.

Models. For the selection of LLMs, we chose six current state-of-the-art models for detection experiments. GPT-4o, GPT-4-1106-preview, and GPT-4-0409 are the latest versions from OpenAI with powerful natural language processing capabilities; Claude-3.5-Sonnet is Anthropic's new-generation model focused on safety and interpretability; Gemini-1.5-Pro is Google's high-performance model optimized for multitasking; Llama-3.1-70b-Instruct is Meta's large-scale model specializing in instruction following and generating high-quality text.

Evaluation Metrics. We carry out experiments to answer the following research questions: **RQ1:** How effective is SCALM in detecting bad practices in smart contracts? How do different LLMs affect SCALM? **RQ2:** Can SCALM find bad practices undetectable by other tools? How does it compare with existing tools? **RQ3:** Can SCALM achieve the same effect without including RAG?

4.2 RQ1: Bad Practice Detection

To assess the effectiveness of SCALM in detecting bad practices within smart contracts, we conduct a comprehensive evaluation using a variety of LLMs. The primary objective is to determine how well SCALM identifies known bad practices, as classified by the Smart SWC registry. We also seek to understand the impact of different LLMs on the performance of SCALM in detecting these bad practices.

We use the DAppSCAN dataset containing 23637 smart contracts as the knowledge base to test the detection ability of SCALM. Specifically, we focus on contracts with known SWC vulnerabilities. Our experiments involve running SCALM with multiple LLM configurations, including GPT-4o, GPT-4-1106-preview, GPT-4-0409, Claude-3.5-Sonnet, Gemini-1.5-Pro, and Llama-3.1-70b-Instruct.

For each model, we evaluate the framework's ability to identify instances of bad practices across 35 SWC categories correctly. Each detection instance is recorded as a success or failure based on whether the LLM accurately identifies the bad practice by its SWC-ID or keywords.

The results of the experiments are summarized in Table 2. The table shows the detection capabilities of each LLM across the different SWC categories, with a checkmark (✓) indicating successful detection and a cross (✗) indicating a failure.

Among the LLMs evaluated, GPT-4o demonstrates the highest detection accuracy, successfully identifying bad practices across 33 of the 36 SWC categories. This model is particularly effective in detecting vulnerabilities such as *Integer Overflow and Underflow* (SWC-101), *Outdated Compiler Version* (SWC-102), and *Reentrancy* (SWC-107).

SWC-ID	Title	Models					
		GPT-4o	GPT-4-1106	GPT-4-0409	Claude	Gemini	Llama
SWC-100	Function Default Visibility	✓	✗	✗	✗	✗	✗
SWC-101	Integer Overflow and Underflow	✓	✓	✓	✓	✓	✓
SWC-102	Outdated Compiler Version	✓	✓	✓	✓	✓	✓
SWC-103	Floating Pragma	✓	✓	✓	✓	✓	✓
SWC-104	Unchecked Call Return Value	✓	✓	✓	✓	✓	✓
SWC-105	Unprotected Ether Withdrawal	✓	✓	✓	✓	✓	✓
SWC-106	Unprotected SELFDESTRUCT Instruction	✓	✓	✓	✓	✓	✓
SWC-107	Reentrancy	✓	✓	✓	✓	✓	✓
SWC-108	State Variable Default Visibility	✓	✓	✓	✓	✓	✓
SWC-109	Uninitialized Storage Pointer	✗	✗	✗	✗	✗	✗
SWC-110	Assert Violation	✓	✓	✓	✓	✓	✓
SWC-111	Use of Deprecated Solidity Functions	✓	✓	✓	✓	✓	✓
SWC-112	Delegatecall to Untrusted Callee	✓	✓	✓	✓	✗	✗
SWC-113	DoS with Failed Call	✓	✓	✓	✓	✗	✓
SWC-114	Transaction Order Dependence	✗	✗	✗	✗	✓	✗
SWC-115	Authorization through tx.origin	✓	✓	✓	✓	✓	✓
SWC-116	Block values as a proxy for time	✓	✓	✓	✓	✓	✓
SWC-117	Signature Malleability	✓	✓	✓	✗	✗	✓
SWC-118	Incorrect Constructor Name	✓	✗	✓	✓	✗	✗
SWC-119	Shadowing State Variables	✓	✓	✓	✓	✓	✓
SWC-120	Weak Sources of Randomness from Chain Attributes	✓	✓	✓	✓	✓	✓
SWC-123	Requirement Violation	✗	✓	✗	✓	✓	✓
SWC-124	Write to Arbitrary Storage Location	✓	✗	✓	✗	✗	✗
SWC-125	Incorrect Inheritance Order	✗	✗	✗	✗	✗	✗
SWC-126	Insufficient Gas Griefing	✓	✓	✓	✗	✗	✗
SWC-127	Arbitrary Jump with Function Type Variable	✓	✓	✓	✓	✓	✓
SWC-128	DoS With Block Gas Limit	✓	✓	✓	✓	✓	✓
SWC-129	Typographical Error	✓	✓	✓	✓	✓	✓
SWC-130	Right-To-Left-Override control character (U+202E)	✗	✗	✗	✗	✓	✓
SWC-131	Presence of unused variables	✓	✓	✓	✗	✓	✗
SWC-132	Unexpected Ether balance	✓	✓	✓	✓	✗	✗
SWC-133	Hash Collisions With Multiple Variable Length Arguments	✗	✗	✗	✗	✗	✗
SWC-134	Message call with hardcoded gas amount	✓	✓	✓	✓	✓	✓
SWC-135	Code With No Effects	✓	✓	✓	✓	✓	✓
SWC-136	Unencrypted Private Data On-Chain	✗	✗	✗	✓	✗	✗

Table 2: SWC bad practice detection. Full model names are GPT-4o, GPT-4-1106-preview, GPT-4-0409, Claude-3.5-Sonnet, Gemini-1.5-Pro, and Llama-3.1-70b-Instruct. A checkmark (✓) indicates successful detection and a cross (✗) indicates a failure.

However, the performance varies significantly among the different LLMs. For instance, while models like GPT-4-1106-preview and GPT-4-0409 also perform well, they exhibit some inconsistencies, such as missing detections in categories like *Function Default Visibility* (SWC-100).

On the other hand, LLMs like Claude-3.5-Sonnet, Gemini-1.5-Pro, and Llama-3.1-70b-Instruct have a more mixed performance, with successful detections in specific categories but noticeable gaps in others. For example, Gemini-1.5-Pro fails to detect *Delegatecall to Untrusted Callee* (SWC-112) and *Transaction Order Dependence* (SWC-114), while Claude-3.5-Sonnet misses issues related to *Signature Malleability* (SWC-117) and *Incorrect Constructor Name* (SWC-118).

The variations in detection accuracy across different LLMs highlight the importance of model selection in the SCALM framework. The results suggest that while models like GPT-4o can handle a wide range of bad practices effectively, others may require further fine-tuning or additional contextual information to improve their performance.

Additionally, the findings emphasize the need for a robust and diverse LLM ensemble within SCALM to ensure comprehensive coverage of all potential bad practices in smart contracts. By leveraging the strengths of different models, SCALM can achieve a more reliable and thorough detection process, ultimately leading to higher-quality audit reports.

Answer to RQ1. SCALM is a powerful framework for detecting bad practices in smart contracts, especially when supported by advanced LLMs like GPT-4o. However, the choice of LLM significantly impacts the system's overall effectiveness, indicating that ongoing model improvements and updates are essential for maintaining high detection accuracy.

4.3 RQ2: Comparison Experiments

We conduct a series of comparison experiments to address RQ2, which examines whether SCALM can identify bad practices that other tools cannot detect and how it compares with existing tools. These experiments focus on several critical SWC categories, specifically SWC-101 (*Integer Overflow and Underflow*), SWC-104 (*Unchecked Call Re-*

SWC-ID	Tools	Acc(%)	Recall(%)	F1(%)
SWC-101	conkas	49.27	67.20	59.10
	mythril	48.03	16.67	25.70
	oyente	62.33	69.35	66.03
	confuzzius	50.26	10.55	18.03
	SCALM	**95.50**	**94.50**	**95.45**
SWC-104	conkas	59.88	20.12	33.00
	mythril	56.89	16.28	28.00
	oyente	—	—	—
	confuzzius	52.32	12.17	20.81
	SCALM	**98.25**	**99.50**	**98.27**
SWC-107	conkas	71.79	94.51	77.50
	mythril	68.12	62.94	69.26
	oyente	59.12	14.04	24.49
	confuzzius	40.00	1.05	2.04
	SCALM	**95.00**	**94.50**	**94.97**
SWC-112	conkas	—	—	—
	mythril	86.67	54.29	70.37
	oyente	—	—	—
	confuzzius	78.86	7.14	13.33
	SCALM	**98.30**	**95.74**	**97.30**
SWC-116	conkas	89.35	86.99	88.50
	mythril	76.45	50.41	63.87
	oyente	48.20	3.16	6.03
	confuzzius	—	—	—
	SCALM	**93.00**	**88.50**	**92.67**

Table 3: Comparison Experiments. SWC-101 is Integer Overflow and Underflow, SWC-104 is Unchecked Call Return Value, SWC-107 is Reentrancy, SWC-112 is Delegatecall to Untrusted Callee, and SWC-116 uses Block Values as a Proxy for Time.

turn Value), SWC-107 (*Reentrancy*), SWC-112 (*Delegatecall to Untrusted Callee*), and SWC-116 (*Block Values as a Proxy for Time*). We select 1,894 smart contracts from the SmartBugs dataset containing these five categories of bad practices.

Additionally, we collect a set of smart contract defect detection tools from reputable journals and conferences in software and security (e.g., CCS and ASE) as well as Mythril (Mythril 2024), recommended by the official Ethereum community. For comparative analysis, we choose four benchmark smart contract detection tools, namely Mythril, Oyente (Luu et al. 2016), Confuzzius (Torres et al. 2021), and Conkas (Veloso 2024). Several factors are considered in the selection of the tools: (1) the accessibility of the tool's source code; (2) the tool's ability to detect the five categories of bad practices we select; (3) the tool's support for source code written in Solidity; (4) the tool's ability to report the exact location of potentially defective code for manual review.

In these experiments, SCALM is powered by GPT-4o, one of the most advanced LLMs. For each SWC category, we evaluate the performance of SCALM and the other tools based on three key metrics: Accuracy (Acc), Recall, and F1 Score. These metrics provide a comprehensive view of each tool's detection capabilities:

- **Accuracy (Acc)** measures the proportion of correctly identified instances (both true positives and true negatives) out of the total instances.
- **Recall** measures the proportion of actual positive instances correctly identified by the tool.
- **F1 Score** is the harmonic mean of Precision and Recall, providing a single metric that balances false positives and false negatives.

The results of these experiments are summarized in Table 3. The results indicate that SCALM outperforms other tools across all evaluated SWC categories. For instance, in SWC-101 (*Integer Overflow and Underflow*), SCALM achieves an F1 score of 95.45%, significantly higher than Oyente's 66.03%. Similarly, in SWC-104 (*Unchecked Call Return Value*), SCALM reaches an F1 score of 98.27%, while Conkas has only 33.00%. In SWC-107 (*Reentrancy*), SCALM scores 94.97%, compared to Conkas's 77.50%. For SWC-112 (*Delegatecall to Untrusted Callee*), SCALM achieves an F1 score of 97.30%, whereas Mythril manages 70.37%. Lastly, in SWC-116 (*Block Values as a Proxy for Time*), SCALM has an F1 score of 92.67%, outperforming Conkas's 88.50%. These results highlight SCALM 's superior Accuracy and Recall in detecting a range of smart contract bad practices.

The comparison experiments demonstrate that SCALM effectively detects bad practices in smart contracts, outperforming existing tools across multiple SWC categories. This suggests that SCALM can provide a more comprehensive and accurate analysis of smart contract security.

Answer to RQ2. SCALM detects bad practices more accurately than existing tools and identifies bad practices that others often miss. Across the five SWC categories tested, SCALM consistently achieves higher Accuracy, Recall, and F1 scores.

4.4 RQ3: Ablation Experiments

We conduct ablation experiments to address RQ3, investigating whether SCALM can achieve the same performance without including the RAG component. In these experiments, SCALM is also powered by GPT-4o. We compare SCALM with and without RAG across several SWC categories: SWC-101, SWC-104, SWC-107, SWC-112, and SWC-116. The results are summarized in Table 4.

The ablation experiments reveal that including RAG significantly enhances SCALM 's performance across all SWC categories. For SWC-101 (*Integer Overflow and Underflow*), SCALM with RAG achieves an F1 score of 95.45%, compared to 79.71% without RAG. In SWC-104 (*Unchecked Call Return Value*), the F1 score with RAG is 98.27%, while without RAG it is 79.19%. Similarly, for SWC-107 (*Reentrancy*), the F1 score increases from 75.35% without RAG to 94.97% with RAG. In SWC-112 (*Delegatecall to Untrusted Callee*), SCALM with RAG achieves an F1 score of 97.30%, compared to 73.99% without RAG. Finally, for SWC-116 (*Block Values as a Proxy for Time*), the F1 score with RAG is 92.67%, whereas without RAG it is 83.13%.

SWC-ID	Tools	Acc(%)	Recall(%)	F1(%)
SWC-101	w/o RAG	79.25	81.50	79.71
	SCALM	**95.50**	**94.50**	**95.45**
SWC-104	w/o RAG	77.00	87.50	79.19
	SCALM	**98.25**	**99.50**	**98.27**
SWC-107	w/o RAG	73.50	81.00	75.35
	SCALM	**95.00**	**94.50**	**94.97**
SWC-112	w/o RAG	84.69	68.09	73.99
	SCALM	**98.30**	**95.74**	**97.30**
SWC-116	w/o RAG	82.75	85.00	83.13
	SCALM	**93.00**	**88.50**	**92.67**

Table 4: Ablation Experiments. 'w/o RAG' indicates excluding the RAG component, while SCALM includes it.

The results demonstrate that the RAG component is crucial for SCALM 's high performance in detecting smart contract vulnerabilities. The significant drop in Accuracy, Recall, and F1 scores when RAG is excluded indicates that retrieving relevant knowledge and context provided by RAG substantially contributes to the model's effectiveness. Without RAG, SCALM struggles to achieve the same level of precision and Recall, underscoring the importance of this component in enhancing the model's overall capability to identify and report bad practices in smart contracts.

Answer to RQ3. The ablation experiments demonstrate that SCALM cannot achieve the same level of performance without the RAG component. Including RAG significantly improves F1 scores, Accuracy, and Recall across all tested SWC categories.

5 Discussion

The experimental results confirm SCALM's effectiveness in detecting bad practices in smart contracts, outperforming existing tools in accuracy, recall, and F1 scores.

The choice of LLM significantly impacts SCALM's performance, with different models yield varying results. This finding emphasizes the need for careful model selection and ongoing improvements to maintain high accuracy. The RAG component is also crucial in SCALM. Ablation experiments showed significant performance drops when RAG was excluded. This highlights its importance in providing contextual information that enhances detection accuracy.

Despite its strengths, SCALM has limitations in understanding blockchain mechanisms and complex interactions. Specifically, SCALM might not fully grasp the state-dependent nature of certain vulnerabilities, which require a deep contextual understanding of how the contract evolves over time.

Future work will focus on improving SCALM by incorporating advanced LLMs and fine-tuning with domain-specific data. Additionally, we plan to improve the RAG generation strategy to provide more prosperous and accurate contextual information. These improvements will enhance the performance of SCALM for smart contract auditing.

6 Related Work

LLMs have been widely applied and validated for their ability to identify and fix vulnerabilities (Napoli and Gatteschi 2023). In the field of smart contract security, various methods based on LLMs have been proposed, and certain effects have been achieved. Firstly, Boi et al. (Boi, Esposito, and Lee 2024) proposed VulnHunt-GPT, a method that uses the GPT-3 to identify common vulnerabilities in OWASP standards smart contracts.s. Similarly, Xia et al. (Xia et al. 2024) introduced a tool called AuditGPT, which utilizes LLM to automatically and comprehensively verify the ERC rules of smart contracts. They break down large, complex audit processes into small, manageable tasks and design prompts for each ERC rule type to improve audit performance. In terms of fuzz testing, Shou et al. (Shou et al. 2024) presented LLM4FUZZ, which employs LLMs to effectively guide fuzz activities in smart contracts and determine their priorities. This approach can enhance test efficiency and coverage compared to traditional fuzz testing methods.

However, the direct use of pre-trained LLMs is no longer sufficient on some occasions, so many studies have chosen to fine-tune LLMs to meet specific needs. For example, Liu et al. (Liu et al. 2024) proposed PropertyGPT, a system for formal verification of smart contracts by retrieving enhanced property generation. This system utilizes the capability of LLM to automatically generate comprehensive properties of smart contracts, including invariants, preconditions, and postconditions, thus improving the security of smart contracts. On the other hand, Storhaug et al. (Storhaug, Li, and Hu 2023) proposed a novel vulnerability-bound decoding method to reduce the amount of vulnerable code generated by the model. They fine-tuned the LLM to include vulnerability tags when generating code and then prohibited these tags during the decoding process to avoid producing vulnerable codes. Finally, Yang et al. (Yang, Man, and Yue 2024) collected a large number of tagged smart contract vulnerabilities and fine-tuned Llama-2-13B for the automatic detection of vulnerabilities in the decentralized finance (DeFi) domain's smart contracts.

In conclusion, while LLMs show great potential for improving smart contract security, the full realization of their potential requires continuous fine-tuning and adaptation of these models in specific contexts.

7 Conclusion

This paper presents the first systematic study of over 35 bad practices in smart contracts. Building on this extensive analysis, we introduced SCALM, a framework based on LLMs for detecting these bad practices in smart contracts. Leveraging the power of LLMs, along with RAG and Step-Back Prompting, SCALM provides a comprehensive and accurate audit of smart contracts. Experiments on datasets such as DAppSCAN and SmartBugs show that SCALM outperforms existing tools, with the RAG component playing a critical role in improving detection accuracy. These findings demonstrate SCALM's potential to enhance smart contract security, offering developers a robust framework to identify and address bad practices.

Acknowledgments

This work is sponsored by the National Natural Science Foundation of China (No.62402146 & 62362021).

References

Boi, B.; Esposito, C.; and Lee, S. 2024. VulnHunt-GPT: A Smart Contract Vulnerabilities Detector Based on OpenAI chatGPT. In *Proceedings of the ACM/SIGAPP Symposium on Applied Computing (SAC)*, 1517–1524.

Chaliasos, S.; Charalambous, M. A.; Zhou, L.; Galanopoulou, R.; Gervais, A.; Mitropoulos, D.; and Livshits, B. 2024. Smart Contract and DeFi Security Tools: Do They Meet the Needs of Practitioners? In *Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE)*, 1–13.

Devlin, J.; Chang, M.-W.; Lee, K.; and Toutanova, K. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. arXiv:1810.04805.

Du, Z.; Qian, Y.; Liu, X.; Ding, M.; Qiu, J.; Yang, Z.; and Tang, J. 2022. GLM: General Language Model Pretraining with Autoregressive Blank Infilling. arXiv:2103.10360.

Durieux, T.; Ferreira, J. F.; Abreu, R.; and Cruz, P. 2020. Empirical Review of Automated Analysis Tools on 47,587 Ethereum Smart Contracts. In *Proceedings of the ACM/IEEE 42nd International conference on software engineering (ICSE)*, 530–541.

Gao, Y.; Xiong, Y.; Gao, X.; Jia, K.; Pan, J.; Bi, Y.; Dai, Y.; Sun, J.; Wang, M.; and Wang, H. 2024. Retrieval-Augmented Generation for Large Language Models: A Survey. arXiv:2312.10997.

Hambardzumyan, S.; Tuli, A.; Ghukasyan, L.; Rahman, F.; Topchyan, H.; Isayan, D.; McQuade, M.; Harutyunyan, M.; Hakobyan, T.; Stranic, I.; and Buniatyan, D. 2022. Deep Lake: A Lakehouse for Deep Learning. arXiv:2209.10785.

Kasneci, E.; Seßler, K.; Küchemann, S.; Bannert, M.; Dementieva, D.; Fischer, F.; Gasser, U.; Groh, G.; Günnemann, S.; Hüllermeier, E.; et al. 2023. ChatGPT for good? On opportunities and challenges of large language models for education. *Learning and individual differences*, 103: 102274–102282.

Liu, Y.; Xue, Y.; Wu, D.; Sun, Y.; Li, Y.; Shi, M.; and Liu, Y. 2024. PropertyGPT: LLM-driven Formal Verification of Smart Contracts through Retrieval-Augmented Property Generation. arXiv:2405.02580.

Luu, L.; Chu, D.-H.; Olickel, H.; Saxena, P.; and Hobor, A. 2016. Making Smart Contracts Smarter. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 254–269.

Mythril. 2024. Mythril. https://mythril-classic.readthedocs.io/. Accessed: 2024-12-24.

Napoli, E. A.; and Gatteschi, V. 2023. Evaluating ChatGPT for Smart Contracts Vulnerability Correction. In *Proceedings of the IEEE Annual Computers, Software, and Applications Conference (COMPSAC)*, 1828–1833.

Sharma, P.; Jindal, R.; and Borah, M. D. 2023. A Review of Smart Contract-Based Platforms, Applications, and Challenges. *Cluster Computing*, 26(1): 395–421.

Shou, C.; Liu, J.; Lu, D.; and Sen, K. 2024. LLM4Fuzz: Guided Fuzzing of Smart Contracts with Large Language Models. arXiv:2401.11108.

Storhaug, A.; Li, J.; and Hu, T. 2023. Efficient Avoidance of Vulnerabilities in Auto-completed Smart Contract Code Using Vulnerability-constrained Decoding. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 683–693.

Torres, C. F.; Iannillo, A. K.; Gervais, A.; and State, R. 2021. ConFuzzius: A Data Dependency-Aware Hybrid Fuzzer for Smart Contracts. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroS&P)*, 103–119.

Veloso, N. 2024. Conkas: A Modular and Static Analysis Tool for Ethereum Bytecode. https://github.com/nveloso/conkas/. Accessed: 2024-12-24.

Wagner, G. 2018. Eip-1470: Smart contract weakness classification (SWC). https://github.com/ethereum/EIPs. Accessed: 2024-12-24.

Xia, S.; Shao, S.; He, M.; Yu, T.; Song, L.; and Zhang, Y. 2024. AuditGPT: Auditing Smart Contracts with ChatGPT. arXiv:2404.04306.

Yang, Z.; Man, G.; and Yue, S. 2024. Automated Smart Contract Vulnerability Detection Using Fine-tuned Large Language Models. In *Proceedings of the International Conference on Blockchain Technology and Applications (ICBTA)*, 19–23.

Zheng, H. S.; Mishra, S.; Chen, X.; Cheng, H.-T.; Chi, E. H.; Le, Q. V.; and Zhou, D. 2024a. Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models. arXiv:2310.06117.

Zheng, Z.; Su, J.; Chen, J.; Lo, D.; Zhong, Z.; and Ye, M. 2024b. Dappscan: building large-scale datasets for smart contract weaknesses in dapp projects. *IEEE Transactions on Software Engineering*, 50(6): 1360–1373.

Zone, S. 2024. SlowMist Hacked. https://hacked.slowmist.io. Accessed: 2024-12-24.