SCLA: Automated Smart Contract Summarization via LLMs and Control Flow Prompt

Anonymous ACL submission

Abstract

Smart contract code summarization is crucial for efficient maintenance and vulnerability mitigation. While many studies use Large Language Models (LLMs) for summarization, their performance still falls short compared to fine-tuned models like CodeT5+ and CodeBERT. Some approaches combine LLMs with data flow analysis but fail to fully capture the hierarchy and control structures of the code, leading to information loss and degraded summarization quality. We propose SCLA, a multimodal LLMs-based method that enhances summarization by integrating a Function Call Graph (FCG) and semantic facts from the code's control flow into a semantically enriched prompt. SCLA uses a control flow extraction algorithm to derive control flows from semantic nodes in the Abstract Syntax Tree (AST) and constructs the corresponding FCG. Code semantic facts refer to both explicit and implicit information within the AST that is relevant to smart contracts. This method enables LLMs to better capture the structural and contextual dependencies of the code. We validate the effectiveness of SCLA through comprehensive experiments on a dataset of 40,000 real-world smart contracts. The experiment shows that SCLA significantly improves summarization quality, outperforming the SOTA baselines with improvements of 26.7%, 23.2%, 16.7%, and 14.7% in BLEU-4, METEOR, ROUGE-L, and BLEURT scores, respectively.

1 Introduction

Smart contracts (Liao et al., 2023) are self-executing programs on Ethereum, and the blockchain's immutability complicates vulnerability maintenance (Zhang et al., 2022). Solidity, designed specifically for smart contract development, compiles code into bytecode and ABI for execution on the Ethereum Virtual Machine (EVM). Unlike general-purpose languages like Java and Python, Solidity emphasizes security with strict type safety and single-threaded execution. Analyzing Solidity code requires examining syntax, semantics, and state management. Even minor vulnerabilities can result in financial losses (Kushwaha et al., 2022), making smart contract code summarization essential for improving efficiency and reducing security risks. Smart contract summarization has received less attention than Java and Python, with traditional methods relying on deep learning and fine-tuning. Yang et al. (Yang et al., 2021) proposed MMTrans, integrating deep learning with structure-based traversal (SBT) and Abstract Syntax Tree (AST) graphs for code summarization. Lei et al. (Lei et al., 2024) introduced FMCF, a Transformer-based method that fuses multi-scale features to preserve both semantic and syntactic information. Zhao et al. (Zhao et al., 2024) proposed SCCLLM, combining context learning with information retrieval to improve summarization.

However, fine-tuned models are often limited by the quality and scale of their training data, which adversely affects their performance. Additionally, these models are prone to knowledge forgetting (De Lange et al., 2022), reducing their adaptability to emerging or evolving code patterns. In contrast, large language models (LLMs) exhibit stronger generalization capabilities through pre-training on large-scale and diverse datasets, often outperforming traditional fine-tuned models. Nevertheless, existing LLM-based approaches generally focus on isolated function-level code snippets, neglecting the contextual role that these functions play within the entire smart contract. This limitation hinders LLMs' ability to fully capture the semantic context of functions. Ahmed et al. (Ahmed et al., 2024) also highlight that LLMs struggle with implicit semantics, which frequently leads to the omission of critical information. Although LLMs demonstrate superior generalization, fine-tuned models (e.g., CodeBERT (Feng et al., 2020) and CodeT5 (Wang et al., 2021)) still achieve better performance in smart contract code summarization tasks, particularly with respect to the semantic conciseness and descriptive accuracy of generated summarization (Wang et al., 2023). Therefore, how to effectively leverage the powerful capabilities and scalability of LLMs to enhance their performance in smart contract summarization—so that the quality of their outputs surpasses that of fine-tuned models—remains a valuable and promising direction for future research.

To address the limitations of existing methods and generate more secure smart contract code summarization, we propose SCLA (Smart Contract summarization with multimodal LLMs and Semantic Augmentation). SCLA integrates multimodal large language models with control flow analysis to enhance scalability and summarization quality. By incorporating control flow–based semantic information and function call graphs (see Section 3.1), SCLA improves the security and semantic accuracy of summaries generated by large language models. This approach extracts such information through semantic analysis and generates function call graphs. In contrast, CP-BCS (Ye et al., 2023) improves binary code summarization quality by combining control flow graphs with pseudo-code representations. Unlike CP-BCS, which focuses on low-level binary code, SCLA's algorithm is specifically designed for smart contract code. It integrates function call graphs and semantic facts, combined with multimodal large language models, achieving greater flexibility and scalability—thus better meeting the semantic accuracy requirements of smart contract code. To select appropriate few-shot examples for a given code snippet, SCLA employs a fine-tuned Sentence-Transformer (Reimers and Gurevych, 2020) to retrieve semantically similar samples, which are then used to construct task-specific prompts. Its core component, **SemFlow**, is responsible for extracting function call graphs and semantic details. To accurately capture function call relationships within the unique syntactic structure of smart contracts, we designed a dedicated extraction algorithm optimized for smart contract syntax (see Section 3.3). To avoid overloading large language models, non-control flow information is presented separately, while function call graphs are provided in a tagged Portable Network Graphics (PNG) format. We conducted experiments on 14,789 method-comment pairs selected from a GitHub repository containing 40,000 smart contracts, demonstrating that the inclusion of function call graphs enhances the performance of large language models.

Our main contributions can be summarized as follows:

- We propose SCLA, the first framework that integrates multimodal LLMs with smart contract code summarization using control flow prompts. It extracts function call graphs and associated semantic information from the AST, enhancing the LLM's understanding of code structure.

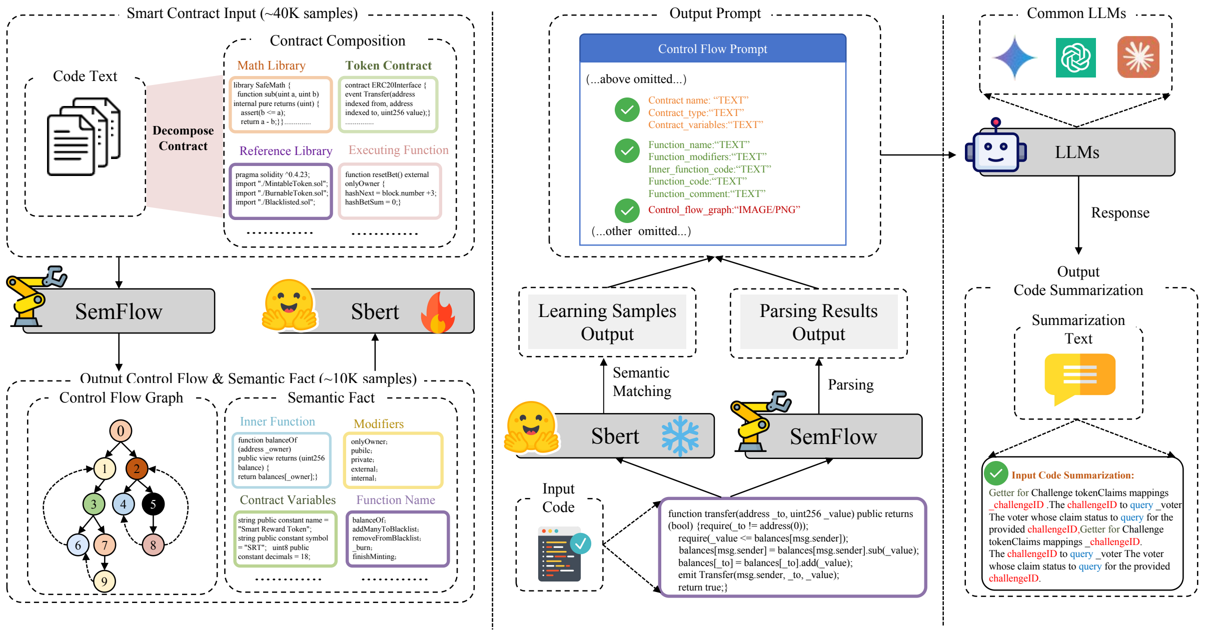
- We conduct extensive experiments on a dataset with 14,795 method-comment pairs, using BLEU-4, METEOR, ROUGE-L, and BLEURT as evaluation metrics. We perform a comparative analysis with state-of-the-art approaches, achieving a 37.53 BLEU-4 score, 52.54 METEOR score, 56.97 ROUGE-L score, and 63.4 BLEURT score.

- We thoroughly evaluated the generalizability of SCLA through extensive experiments on Java and Python datasets, offering valuable insights for future research on control flow-based prompts in other code domains.

2 Related Work

Smart Contract Summarization

Deep learning models have made significant advances in smart contract code summarization. Yang et al. (Yang et al., 2021) proposed MMTrans, which extracts SBT sequences and AST-based graphs to capture global and local semantics using dual encoders and a joint decoder. Transformer models like CodeT5 (Wang et al., 2021) and CodeBERT (Feng et al., 2020) also enhance summarization quality but require extensive fine-tuning and large datasets. LLMs, such as GPT-4o and Gemini-1.5-Pro, excel in few-shot or zero-shot summarization tasks, bypassing fine-tuning. Previous studies (Ahmed and Devanbu, 2023; Ahmed et al., 2024) highlight the benefits of few-shot learning. However, LLMs often produce suboptimal summarization, lacking conciseness and functional generalization. Ahmed et al. (Ahmed et al., 2024) proposed ASAP, incorporating data flow and GitHub context. Still, it fails to capture function call relationships and control flow, suggesting the need for improved semantic facts and control flow integration for better summarization.



```
library SafeMath {
  function sub(uint a, uint b)
internal pure returns (uint) {
   assert(b <= a);
   return a - b;}}..............
```

```
contract ERC20Interface {
event Transfer(address
indexed from, address
indexed to, uint256 value);}
...............
```

```
pragma solidity ^0.4.23;
import "./MintableToken.sol";
import "./BurnableToken.sol";
import "./Blacklisted.sol";
```

```
function resetBet() external
onlyOwner {
hashNext = block.number +3;
hashBetSum = 0;}
```

```
function balanceOf
(address _owner)
public view returns (uint256
balance) {
return balances[_owner];}
```

```
onlyOwner;
pubilc;
private;
external;
internal;
```

```
string public constant name =
"Smart Reward Token";
string public constant symbol
= "SRT";   uint8 public
constant decimals = 18;
```

```
balanceOf;
addManyToBlacklist;
removeFromBlacklist;
_burn;
finishMinting;
```

(...above omitted...)

Contract name: "TEXT"
Contract_type:"TEXT"
Contract_variables:"TEXT"

Function_name:"TEXT"
Function_modifiers:"TEXT"
Inner_function_code:"TEXT"
Function_code:"TEXT"
Function_comment:"TEXT"

Control_flow_graph:"IMAGE/PNG"

(...other omitted...)

```
function transfer(address _to, uint256 _value) public returns
(bool) {require(_to != address(0));
   require(_value <= balances[msg.sender]);
   balances[msg.sender] = balances[msg.sender].sub(_value);
   balances[_to] = balances[_to].add(_value);
   emit Transfer(msg.sender, _to, _value);
   return true;}
```

Getter for Challenge tokenClaims mappings _challengeID .The challengeID to query _voter The voter whose claim status to query for the provided challengeID,Getter for Challenge tokenClaims mappings _challengeID.
The challengeID to query _voter The voter whose claim status to query for the provided challengeID.

(a) Semantic Extraction Phase **(b) Prompt Construction Phase** **(c) LLMs Inference**

Figure 1: Overview of our proposed framework, SCLA, powered by Google's Gemini-1.5-Pro, performs automated generation of smart contract code summarization. SCLA extracts control flow semantic facts from smart contract code and uses Gemini-1.5-Pro to generate code summarization from control flow semantic facts.

3 METHODOLOGY

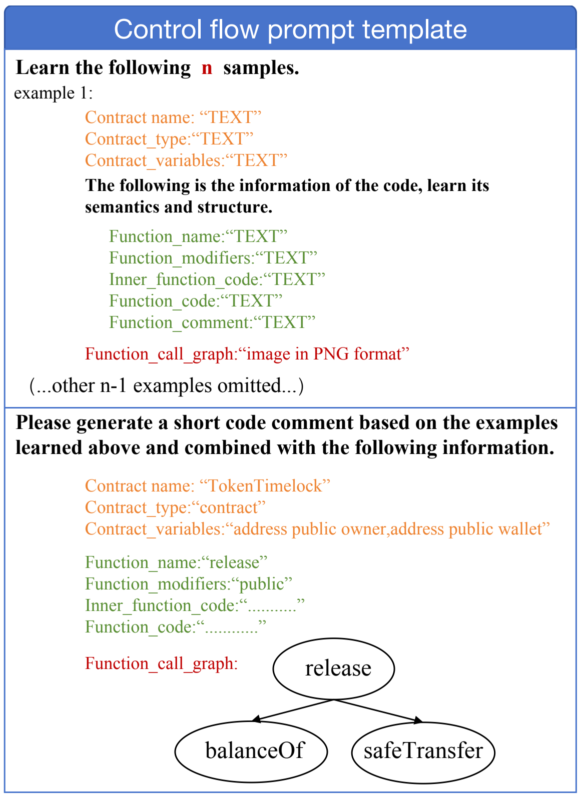
3.1 Control Flow Prompt

In this section, we discuss the control flow prompt and the corresponding semantic facts utilized by SCLA, as illustrated in Figure 2. These semantic facts are carefully integrated into the prompts to enhance the LLMs' ability to generate more accurate, relevant, and comprehensive summarization. The appendix Section B shows more detailed information about the control flow prompt.

Function Call Graph & Inner Function. We define the set of inner functions as those invoked within the target function, with each element referred to as an inner function. The function call graph captures the precise sequence of function calls, representing the control flow of the target code. This graph is used as control flow input for the LLMs, along with the set of inner functions, to provide valuable additional context about invoked functions. This approach mitigates misinterpretation based solely on function names, significantly enhancing semantic inference. Moreover, the function call graph helps the LLMs accurately determine the sequence and depth of function calls, thereby aiding in the understanding of complex functions and their interdependencies.

Identifiers. Previous studies highlight that identifiers play a critical role in helping language models retrieve valuable information for code summarization (Ahmed and Devanbu, 2022). Identifiers, including modifiers, local variables, and function names, offer essential context about the code's operations. By understanding an identifier's role, the language model can better interpret the code. In our approach, we perform a deep traversal of the function's AST, visiting each AST node to collect identifiers along with their corresponding roles, and incorporate this information into the prompt.

Contract Name & Global Member Variables. Incorporating domain-specific information into prompts greatly enhances LLMs' overall performance and effectiveness, particularly in specialized tasks such as smart contract analysis. For instance, smart contract names (Kong et al., 2024) often reflect their functional roles or token names (Chen et al., 2021), providing valuable contextual information for the LLMs. Additionally, global member variables, such as contract addresses and account balances, assist LLMs in more effectively understanding contract functions and their interrelations. This significantly reduces the need for LLMs to infer complex operations from variable names, leading to more precise descriptions and significantly improved summarization accuracy.


Control flow prompt template

Learn the following n samples.

example 1:

Contract name: “TEXT”
Contract_type:“TEXT”
Contract_variables:“TEXT”

The following is the information of the code, learn its semantics and structure.

Function_name:“TEXT”
Function_modifiers:“TEXT”
Inner_function_code:“TEXT”
Function_code:“TEXT”
Function_comment:“TEXT”

Function_call_graph:“image in PNG format”

(...other n-1 examples omitted...)

Please generate a short code comment based on the examples learned above and combined with the following information.

Contract name: “TokenTimelock”
Contract_type:“contract”
Contract_variables:“address public owner,address public wallet”

Function_name:“release”
Function_modifiers:“public”
Inner_function_code:“..........”
Function_code:“...........”

Function_call_graph:

Figure 2: An Example of Control Flow Prompt.

3.2 Semantic-based Retrieval

In this paper, we use the **Sentence-Transformer** (SBERT) model (Reimers and Gurevych, 2020) to semantically match the identified code samples in the repository that are most similar to the target code snippet, which are then used as few-shot learning examples in the prompt. We selected SBERT because of its superior language understanding capabilities compared to the CCGIR (Yang et al., 2022) method used in SCCLLM (Zhao et al., 2024). SBERT, fine-tuned based on BERT and built upon the Transformer architecture, demonstrates excellent performance in semantic similarity tasks. In contrast, CCGIR relies on Code Clone Graphs (Zou et al., 2020) (CCG), which only capture structural and lexical features, limiting its ability to represent deeper semantics. First, we partition the samples in the repository into training, test, and validation sets (as shown in Table 1) and fine-tune the SBERT model using the training set. We begin by vectorizing the given sentences S_1 (smart contract code) and S_2,S_3 (human-written comments), as described by the following formula:

$$\begin{aligned}\mathbf{v}_1 &= \text{Pooling}(BERT(S_1))\\ \mathbf{v}_2 &= \text{Pooling}(BERT(S_2))\end{aligned} \tag{1}$$

Sbert is trained with contrastive or triplet loss to bring similar sentences closer and push dissimilar ones apart in the embedding space. Given positive pair (S_1, S_2) and negative pair (S_1, S_3), it optimizes this loss:

$$\begin{aligned}\mathcal{L} = \max\,(0, &\text{cosine\_similarity}(\mathbf{v}_1, \mathbf{v}_2)\\ &-\text{cosine\_similarity}(\mathbf{v}_1, \mathbf{v}_3) + \Delta)\end{aligned} \tag{2}$$

where Δ is a margin hyperparameter that controls the minimum desired similarity difference, and $\mathbf{v}_1$ and $\mathbf{v}_2$ are the vector representations of sentences S_1 and S_2.

Finally, we compute the cosine similarity between the target code vector and the repository code vectors to identify the most semantically similar samples. The formula is as follows:

$$\text{cosine\_similarity}(\mathbf{v}_1, \mathbf{v}_2) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{\|\mathbf{v}_1\|\|\mathbf{v}_2\|} \tag{3}$$

For each target sample, we rank the repository samples by cosine similarity in descending order. The top k matches, as specified by the parameter number_top_matches, are selected and stored in a result dictionary, which contains the matched code snippets and their similarity scores. If a file path is provided, the results are serialized and saved in JSON format for further analysis or review.

3.3 SCLA Framework

Figure 1 illustrates the overall framework of SCLA. We outline the three stages of the SCLA process for generating smart contract code summarization.

Semantic Extraction: We split the .sol files based on the "contract node" in the AST of the smart contract code. This method enables us to split the contents of the .sol files into individual smart contracts, thereby avoiding parsing errors. Subsequently, we extract the code and comments of each smart contract using regular expressions, which are then passed to SemFlow for semantic extraction. The function call graphs and semantic facts are stored in a repository, indexed by the contract file path and named using UUIDs.

Prompt Construction: SCLA uses few-shot learning to enhance LLMs’ code summarization performance. Sentence-Transformer (Reimers and Gurevych, 2020) retrieves the top k semantically similar code samples. The extracted semantic information, including function call graphs, function arguments, function modifiers, and contract metadata, is integrated into the prompt.

LLMs Inference: The semantically enhanced prompt, including the function call graph, is input into the LLMs interface to improve understanding of the function call sequence, resulting in higher-quality code summarization.

Algorithm 1 Source Data to Function Call Tree

```
1: Input: Source code $f$ to be parsed by Solparser; initialized empty dictionary $T$
2: Output: Function call tree $T$
3: $AST \leftarrow$ Solparser.parser($f$)
4: $T \leftarrow \{\}$
5: for each $c$ in $AST$ do
6:     for each $g$ in $c$ do
7:         for each $x$ in $g$.calls do
8:             $n \leftarrow x$.name
9:             if $n \notin T[c][g]$ then
10:                $T[c][g][n] \leftarrow \{c : c$, count : $1\}$
11:            else
12:                $T[c][g][n]$.count $\leftarrow T[c][g][n]$.count $+ 1$
13:            end if
14:        end for
15:    end for
16: end for
17: for each $c$, $g$ in $T$ do
18:     CreateCallTree($c$, $g$, $T[c][g]$, $T$)
19: end for
20: return $T$
21: function CREATECALLTREE(p, k, n, T)
22:     for each $m$ in $n$.keys do
23:         $o \leftarrow n[m]$
24:         if $m \notin T[p][k]$.keys then
25:             $T[p][k][m] \leftarrow T[o.\mathrm{c}][m]$
26:             CreateCallTree($k$, $m$, $T[o.\mathrm{c}][m]$, $T$)
27:         end if
28:     end for
29: end function
```

3.4 Control Flow Extraction

We use **SemFlow**, a component integrated with a control flow extraction algorithm, to extract function call graphs from the AST as control flow input in the prompt." The algorithm in 1 demonstrates the entire extraction process. It first uses an AST parsing tool to parse the input code into an AST. The AST is traversed in a depth-first manner to remove irrelevant nodes, such as imports. Function nodes with calls are marked in the "FunctionCall" field, allowing the construction of a reference tree (lines 5-20 of Algorithm 1). The depth of the reference tree ranges from 2 to 3 layers, depending on the presence of function calls. When a third-level call points to a second or third-level node, the reference tree is transformed into a complete call tree by grafting branch nodes (lines 21-29 of Algorithm 1). The call tree is then visualized using Graphviz and saved to the code sample repository.

Type	Train	Validation	Test
Number	11032	2758	1000
Avg. tokens in codes	42.44	42.08	41.95
Avg. tokens in comments	26.34	26.16	26.66

Table 1: Statistics of Experimental Dataset.

4 EXPERIMENT

In the empirical study, we conducted comparison, ablation, and generalization experiments. First, we used **SemFlow** to process the raw data and generate semantic facts, data flow graphs, and the semantic sample library. The code snippets were then input into SCLA for summarization and evaluation. In the comparison experiment, we varied the number of few-shot learning samples and compared the evaluation scores with baseline methods. Ablation experiments assessed the contribution of different semantic components, while generalization experiments extended SCLA to Java and Python code summarization tasks. The results and expert evaluations validate the effectiveness of SCLA in generating smart contract code summarizations.

4.1 Experiment Settings

All our experiments are performed on a computer equipped with an NVIDIA GeForce RTX 4070Ti GPU (12GB graphic memory), Intel (R) Core (TM) i9-13900K, running Ubuntu 22.04 LTS.

4.2 Dataset

The raw data for this study, provided by Liu et al. (Liu et al., 2021), includes 40,000 smart contracts from Etherscan.io[1], created by professional developers and deployed on Ethereum. Building on Yang et al.'s method (Yang et al., 2021), we used AST location data and regular expressions to segment code and extract functions with comments. Samples with comments under six characters were removed. Manual filtering eliminated low-quality comments, including (1) generic templates; (2) identical comments for different code; (3) incomplete sentences; and (4) ambiguous meanings. After cleaning, 14,790 <method, comment> pairs remained. The dataset is split into 11,032 training, 2,758 validation, and 1,000 test samples. Average token counts appear in Table 1.

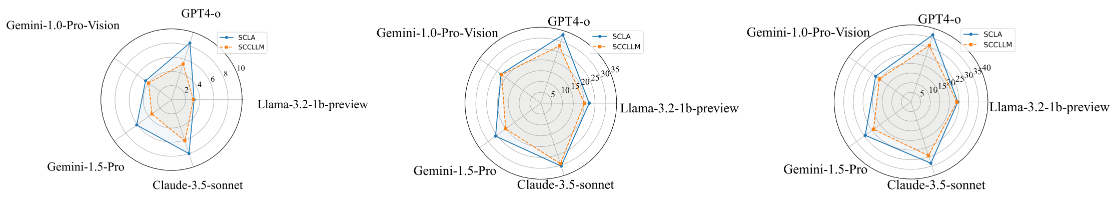
4.3 Baseline

We compare our proposed SCLA with six state-of-the-art methods, including general code summarization models such as **CodeT5** (Wang et al., 2021), **CodeT5+** (Wang et al., 2023), and **CodeBERT** (Feng et al., 2020), deep learning-based smart contract code summarization methods **MMTran** (Yang et al., 2021) and **FMCF** (Lei et al.,

[1]https://etherscan.io/

Model	# of sample	BLEU-4			METEOR			ROUGE-L			p-value
		Zero-Shot	+CFG +IF	Gain(%)	Zero-Shot	+CFG +IF	Gain(%)	Zero-Shot	+CFG +IF	Gain(%)	
Llama-3.2-1b-preview	11032	3.03	**5.43**	+79.21%	19.58	**23.97**	+22.42	18.88	**23.49**	+24.42	<0.01
GPT-4o	11032	5.34	**7.45**	+39.51%	22.32	**26.62**	+19.27	25.32	**32.62**	+28.83	<0.01
Gemini-1.0-Pro-Vision	11032	3.01	**5.32**	+76.74%	16.89	**20.73**	+22.73	18.46	**20.31**	+10.02	<0.01
Gemini-1.5-Pro	11032	3.21	**5.87**	+82.87%	19.89	**25.61**	+28.76	23.95	**27.42**	+14.49	<0.01
Claude-3.5-sonnet	11032	3.31	**5.32**	+60.73%	23.42	**28.62**	+22.20	25.82	**30.12**	+16.65	<0.01

Table 2: Performance of different LLMs on smart contract code summarization, measured using BLEU-4, METEOR, ROUGE-L. p-values are calculated applying a one-sided pairwise Wilcoxon signed-rank test and B-H corrected.



(a) Comparison Results on BELU-4 (b) Comparison Results on METEOR (c) Comparison Results on ROUGE-L

Figure 3: The Comparison of BLEU, METEOR, and ROUGE-L Scores on Our Test Set Under Five Different LLMs, Using the SCCLLM and the Proposed SCLA for Zero-Shot Summarization Tasks.

2024), and smart contract-specific code summarization methods based on the latest LLMs, such as **SCCLLM** (Zhao et al., 2024).

4.4 Performance Metrics

To evaluate SCLA performance against baselines, we adopted various automatic performance metrics, including **BLEU-4** (Papineni et al., 2002), **METEOR** (Banerjee and Lavie, 2005), **ROUGE-L** (Lin, 2004), and **BLEURT** (Sellam et al., 2020). These metrics effectively assess the similarity between the automatically generated smart contract summarization and the real human-generated summarization. BLEURT, in particular, calculates similarity based on sentence semantics by using a pretrained BERT model, providing a more accurate reflection of semantic meaning.

4.5 Main Results

We conducted a comprehensive evaluation of the Gemini-1.5-Pro-powered SCLA under two distinct experimental settings. Gemini-1.5-Pro was selected due to its significantly higher context token capacity compared to Claude-3.5-Sonnet and GPT-4o. This advantage is particularly critical in scenarios where the target function exhibits deep callback chains, leading to large function call graphs that may exceed the context length limits of Claude-3.5-Sonnet and GPT-4o. Moreover, Gemini-1.5-Pro offers a fully free API, making it a more cost-effective choice in high token consumption environments. The SCLA demonstrated substantial performance improvements in smart contract code summarization tasks under both zero-shot and few-shot learning settings. These findings provide valuable insights and contributions to the research community. The specific results are as follows:

Zero-shot Results. To evaluate the impact of function call graphs and internal functions on LLMs-generated code summarization, we conducted experiments using GPT-4o, Gemini-1.5-Pro, and Claude-3.5-Sonnet under zero-shot conditions. The experiment had two phases: first, the target code was embedded into the prompt and evaluated with standard metrics; second, the prompt was enhanced with internal functions and function call graphs, followed by re-evaluation. Table 2 shows that incorporating internal functions and call graphs improved summarization. GPT-4o improved by 39.51%, 19.27%, and 28.83%; Gemini-1.5-Pro by 82.87%, 28.76%, and 14.49%; and Claude-3.5-Sonnet by 60.73%, 22.20%, and 16.65%. However, Gemini-1.5-Pro underperformed compared to GPT-4o and Claude-3.5-Sonnet. **These results validate our hypothesis that function call graphs enhance smart contract summarization**. To further validate the control flow prompt's effectiveness, we compared SCLA with SCCLLM using five multimodal models on the test set. Results in Figure 3 show SCLA outperforming SCCLLM in BLEU, METEOR, and ROUGE-L scores. **This demonstrates that SCLA with the control flow prompt outperforms SCCLLM, confirming the effectiveness of control flow prompts**.

Few-shot Results. To evaluate the performance of SCLA against SOTA baseline models, we conducted a validation experiment. Since SCLA em-

Approach	#of training sample	#of test sample	BLEU-4	METEOR	ROUGE-L	BLEURT	p-values
CodeT5+	11032	1000	28.95	45.62	49.77	57.79	/
CodeT5	11032	1000	27.24	43.31	49.03	52.61	/
CodeBERT	11032	1000	26.31	39.57	44.52	52.74	/
MMTran	11032	1000	22.12	38.92	40.12	54.73	/
FMCF	11032	1000	29.98	36.67	51.21	51.73	/
SCCLLM (One-Shot)	/	1000	19.45	20.12	19.12	36.56	<0.01
SCCLLM (Three-Shot)	/	1000	29.73	35.33	49.44	50.91	<0.01
SCCLLM (Five-Shot)	/	1000	31.73	48.12	60.44	58.74	<0.01
SCLA (Zero-Shot)	/	1000	6.09	25.80	29.45	46.63	<0.01
SCLA (One-Shot)	/	1000	25.46	42.78	47.55	57.07	<0.01
SCLA (Three-Shot)	/	1000	**35.15**	**51.80**	**55.89**	**63.11**	<0.01
SCLA (Five-Shot)	/	1000	**37.53**	**52.54**	**56.97**	**63.44**	<0.01

Table 3: The impact of different few-shot learning quantities on SCLA performance with Gemini-1.5-Pro. p-values are calculated applying a one-sided pairwise Wilcoxon signed-rank test and B-H corrected.

Type	Zero-Shot	One-Shot	Three-Shot	Five-Shot
Avg. tokens in prompt	561.4	1154.8	2242.5	3330.3

Table 4: Number of Tokens Consumed with Different Numbers of Learning Sample for SCLA.

Approach	Prompt Component	BLEU-4	METEOR	ROUGE-L	BLEURT
SCLA	ALL	**6.09**	25.80	**29.45**	**46.63**
	-FCG	5.21	27.77	27.94	45.90
	-IF	4.42	25.43	26.23	44.56
	-Id&MGV	5.62	25.47	29.01	46.32

Table 5: Ablation study. Effect of Semantic Augmentation on Gemini-1.5-Pro Generated Summarization. FCG is Function Call Graph, IF is Inner Function, Id&MGV is Identifiers&Global Member Variables.

ploys few-shot learning, we tested its performance under Zero-Shot, One-Shot, Three-Shot, and Five-Shot conditions to investigate the number of learning samples required for optimal performance. The results (see Table 3) indicate that SCLA initially lags behind the baseline models in Zero-Shot and One-Shot settings. However, starting from Three-Shot, SCLA outperforms the baseline models across all four evaluation metrics: BLEU-4, METEOR, ROUGE-L, and BLEURT. Compared to FMCF, SCLA improved by 17.24%, 41.26%, 9.14%, and 22.00%, and compared to CodeT5+, the improvements were 21.42%, 13.55%, 12.30%, and 9.21%. **Compared to all baseline models, SCLA showed average improvements of 26.7%, 23.2%, 16.7%, and 14.7% in these metrics**. Performance continued to improve under Five-Shot, although the gains were modest. We also analyzed token consumption to determine the optimal number of few-shot samples (see Table 4). The token consumption for Five-Shot was 48.51% higher than for Three-Shot, but the average improvement in generated code summarization metrics was only 2.20%. Therefore, **Three-Shot provides the best balance between performance and efficiency**.

4.6 Ablation Study

We conducted ablation experiments to quantify the impact of individual semantic facts in SCLA on Gemini-1.5-Pro's code summarization under Zero-Shot learning. As shown in Table 5, five variants were tested by selectively removing semantic elements from the enhanced prompts. The results highlight the importance of inner function ordering, function call graphs, identifiers, and global member variables. Notably, removing inner functions caused a performance drop of up to 27.42%, while excluding function call graphs led to significant declines in BLEU-4 (14.45), ROUGE-L (5.13), and BLEURT (1.57). Eliminating identifiers and global variables also reduced performance across all metrics. These results confirm that inline functions and call graphs are essential for improving summarization quality. **Moreover, global member variables help preserve semantic consistency, and function call graphs offer structural context crucial to summarization accuracy**. Removing these components weakens both coherence and completeness, validating the necessity of each semantic element.

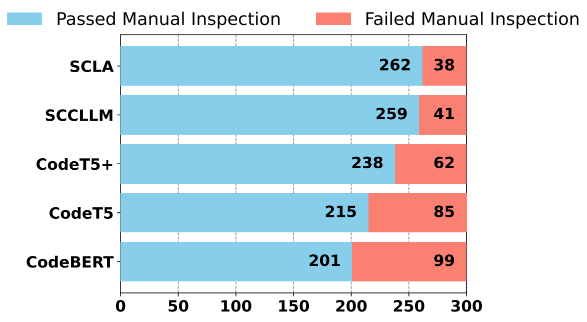


Figure 4: Human Evaluation Results of 300 Code Summarizations Generated by SCLA and the Baseline.

Approach	#of Training Sample	#of Test Sample	Java			Python		
			BLEU-4	METEOR	ROUGE-L	BLEU-4	METEOR	ROUGE-L
CodeBERT	8000	1000	19.91	25.11	34.34	20.56	33.37	33.19
CodeT5	8000	1000	22.45	28.98	41.98	28.82	37.98	39.52
CodeT5+	8000	1000	28.82	39.79	49.31	34.67	46.98	47.34
SCLA	**/**	1000	**34.34**	**50.66**	**60.71**	**37.34**	**52.61**	**57.49**

Table 6: The performance of our proposed method and the baseline model was evaluated on Java and Python datasets.

Language	Model	#of Test Sample	BLEU-4			BLEURT			p-values
			SCCLLM	SCLA	Gain (%)	SCCLLM	SCLA	Gain (%)	
Java	GPT-4o	1000	28.59	**38.43**	+34.42	50.34	**68.89**	+36.85	<0.01
	Gemini-1.5-Pro	1000	23.22	**31.43**	+35.36	56.33	**63.67**	+13.03	<0.01
	Claude-3.5-sonnet	1000	31.05	**39.13**	+26.02	58.89	**70.90**	+20.40	<0.01
Python	GPT-4o	1000	22.78	**29.56**	+29.76	55.90	**64.23**	+14.90	<0.01
	Gemini-1.5-Pro	1000	20.15	**26.06**	+29.33	51.78	**61.03**	+17.86	<0.01
	Claude-3.5-sonnet	1000	25.45	**33.77**	+32.69	58.21	**73.56**	+26.37	<0.01
Overall	/	/	25.21	**33.06**	**+31.14**	55.24	**67.05**	**+21.38**	<0.01

Table 7: The performance of SCLA and SCCLLM on the Java and Python tasks, driven by three different LLMs, was evaluated using BLEU-4 and BLEURT as metrics. To assess the statistical significance of the results, p-values were calculated using a one-sided pairwise Wilcoxon signed-rank test, with Benjamini-Hochberg (B-H) correction applied for multiple comparisons.

4.7 Human Evaluation of Summarization Generated by SCLA and the Baseline

To assess the summarization generated by SCLA, we randomly selected 300 samples from the smart contract code summarization generated by SCLA and baseline models for manual evaluation. This evaluation focused on similarity, conciseness, and completeness, categorizing the summarization as usable or unusable. To reduce subjectivity and bias, six volunteer evaluators, all Chinese graduate students with experience in smart contract development, were recruited and briefed on the research and evaluation standards. The results, shown in Figure 4, reveal that SCLA generated the fewest unusable summarization, outperforming all baseline models. **These findings demonstrate that SCLA is more likely to generate satisfactory smart contract code summarization, reducing the chances of low-quality outputs.**

4.8 Generalization Study

To evaluate the generalization ability of SCLA, we selected 10,000 samples each from Java and Python in the CodeSearchNet (Husain et al., 2019) dataset and randomly sampled 1,000 instances per language as test sets. Since SCLA's FCG extraction algorithm was originally designed for Solidity, we adapted it to accommodate the syntax of Java and Python. Using BLEU-4, METEOR, and ROUGE-L as evaluation metrics, SCLA achieved improvements over CodeT5+ of 19%, 12%, and 23% on the Java dataset, and 7%, 18%, and 21% on the Python dataset, respectively. Furthermore, leveraging GPT-4o, Gemini-1.5-Pro, and Claude-3.5-Sonnet, we compared SCLA with SCCLLM across both datasets using BLEU-4 and BLEURT metrics. The results indicate that SCLA consistently outperforms SCCLLM across all models, with average gains of 31.14 in BLEU-4 and 21.38 in BLEURT. These findings demonstrate that control flow–based prompts exhibit strong generalization to Java and Python, effectively enhancing large language models' understanding of code structure and improving code summarization quality.

5 Conclusion

We propose that function call graphs enhance LLMs' understanding of smart contract code semantics, and experiments confirm their positive impact on code comprehension. Ablation studies assess the contribution of each prompt component to summarization quality. SCLA is a framework that combines LLMs with control flow prompts, outperforming six baseline models. The experiments show that, compared to other baseline models, SCLA significantly improves BLEU-4, METEOR, ROUGE-L, and BLEURT scores with improvements of 30.34%, 23.15%, 16.74%, and 14.86%, respectively. We also extended SCLA to Java and Python code, further improving summarization and providing new insights for advancing LLM-generated code summarization.

Limitations

Our framework enhances Gemini-1.5-Pro's understanding using function call graphs. However, Gemini-1.5-Pro struggles with deep call stacks or circular calls. Figure 5 shows that circular chains, like transferFrom → removeTokenFrom → ownerOf → isApprovedOrOwner → transferFrom, confuse the model, leading to misinterpretations and incorrect summarization. In contrast, Gemini-1.5-Pro handles typical tree structures even with a depth of 5. Further research is needed to explore the impact of loop calls and depth on-call interpretation. We have not yet fully resolved this issue, but we propose an approach whereby the LLM processes the function call graph in a specified order (e.g., top-down) and arranges functions hierarchically. In this structure, normal calls are represented as higher-level functions invoking lower-level ones, while cyclic calls appear as lower-level functions invoking higher-level ones. This hierarchical arrangement helps the LLM avoid misinterpreting the position of cyclic calls.

Another key challenge in using LLMs for smart contract code summarization is the potential exposure of test data during pre-training. Since general-purpose LLMs like GPT-4o and Gemini-1.5-Pro are not publicly accessible, direct verification of this exposure is difficult. Additionally, LLMs' memorization capability can produce artificially high scores if prior summarizations are retained. We also analyzed the effect of few-shot learning on SCLA's performance in Section 3. Our results show that SCLA outperforms the baseline with a Three-Shot setup, while performance gains plateau at five shots, with a 1.5x increase in computational cost.

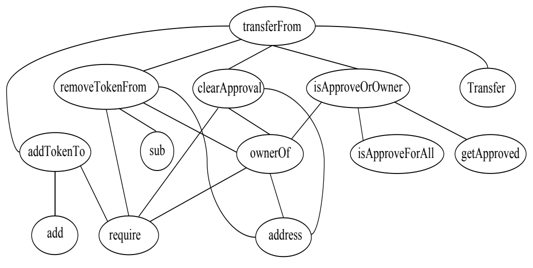


Figure 5: An Example of a Function Call Graph in Which Gemini-1.5-Pro Has Difficulty Understanding the Call Information.

References

Toufique Ahmed and Premkumar Devanbu. 2022. Multilingual training for software engineering. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1443–1455.

Toufique Ahmed and Premkumar Devanbu. 2023. Few-shot training llms for project-specific code-summarization. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–5.

Toufique Ahmed, Kunal Suresh Pai, Premkumar Devanbu, and Earl Barr. 2024. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13.

Satanjeev Banerjee and Alon Lavie. 2005. Meteor: An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Xiangping Chen, Peiyong Liao, Yixin Zhang, Yuan Huang, and Zibin Zheng. 2021. Understanding code reuse in smart contracts. In *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 470–479. IEEE.

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2022. A continual learning survey: Defying forgetting in classification tasks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 3366–3385.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Dechao Kong, Xiaoqi Li, and Wenkai Li. 2024. Characterizing the solana nft ecosystem. In *Companion Proceedings of the ACM on Web Conference*, pages 766–769.

Satpal Singh Kushwaha, Sandeep Joshi, Dilbag Singh, Manjit Kaur, and Heung-No Lee. 2022. Systematic review of security vulnerabilities in ethereum blockchain smart contract. *IEEE Access*, pages 6605–6621.

Gang Lei, Donghua Zhang, Jianmao Xiao, Guodong Fan, Yuanlong Cao, and Zhiyong Feng. 2024. Fmcf:

A fusing multiple code features approach based on transformer for solidity smart contracts source code summarization. *Applied Soft Computing*, page 112238.

Zeqin Liao, Sicheng Hao, Yuhong Nan, and Zibin Zheng. 2023. Smartstate: Detecting state-reverting vulnerabilities in smart contracts via fine-grained state-dependency analysis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 980—-991.

Chin-Yew Lin. 2004. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81.

Zhenguang Liu, Peng Qian, Xiang Wang, Lei Zhu, Qinming He, and Shouling Ji. 2021. Smart contract vulnerability detection: From pure neural network to interpretable graph feature and expert pattern fusion. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, pages 2751–2759.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318.

Nils Reimers and Iryna Gurevych. 2020. Making monolingual sentence embeddings multilingual using knowledge distillation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 4512–4525.

Thibault Sellam, Dipanjan Das, and Ankur Parikh. 2020. BLEURT: Learning robust metrics for text generation. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 7881–7892.

Yue Wang, Hung Le, Akhilesh Gotmare, Nghi Bui, Junnan Li, and Steven Hoi. 2023. CodeT5+: Open code large language models for code understanding and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 1069–1088.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C.H. Hoi. 2021. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708.

Guang Yang, Ke Liu, Xiang Chen, Yanlin Zhou, Chi Yu, and Hao Lin. 2022. Ccgir: Information retrieval-based code comment generation method for smart contracts. *Knowledge-based systems*, 237:107858.

Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, and Miao Zhang. 2021. A multi-modal transformer-based code summarization approach for smart contracts. In *Proceedings of the IEEE/ACM 29th International Conference on Program Comprehension*, pages 1–12.

Tong Ye, Lingfei Wu, Tengfei Ma, Xuhong Zhang, Yangkai Du, Peiyu Liu, Shouling Ji, and Wenhai Wang. 2023. CP-BCS: Binary code summarization guided by control flow graph and pseudo code. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14740–14752, Singapore.

Shenhui Zhang, Wenkai Li, Xiaoqi Li, and Boyi Liu. 2022. Authros: Secure data sharing among robot operating systems based on ethereum. In *Proceedings of the IEEE 22nd International Conference on Software Quality, Reliability and Security*, pages 147–156. IEEE.

Junjie Zhao, Xiang Chen, Guang Yang, and Yiheng Shen. 2024. Automatic smart contract comment generation via large language models and in-context learning. 168.

Yue Zou, Bihuan Ban, Yinxing Xue, and Yun Xu. 2020. Ccgraph: a pdg-based code clone detector with approximate graph matching. In *Proceedings of the 35th IEEE/ACM international conference on automated software engineering*, pages 931–942.

A Case Study

Upon reviewing the results, we found that the SCLA prompt includes crucial information for effective summarization. Table 8 highlights the differences between real-world smart contract abstracts and Summarization generated by CodeBERT, CodeT5, CodeT5+, and SCLA. CodeBERT identifies key terms like "transfer," "ownership," and "address," but lacks clarity, with ambiguous pronoun references and repetition of the transfer concept. CodeT5 captures "onlyOwner" but overlooks broader global semantics, rendering the second sentence redundant. CodeT5+ addresses this limitation with more precise terminology, such as identifying the object as a "data contract." **In contrast, SCLA's Summarization aligns more closely with real-world Summarization, being both more concise and semantically accurate, omitting redundancy for a much clearer, more refined, and contextually precise structure**.

B Control Flow Prompt

Learn the following 1 sample.
Example 1:
Contract name: KahnAirDrop
Contract type: contract
Contract Variables:

```
{"owner": "address public owner;"},
{"wallet": "address public wallet;"},
{"mineth": "uint256 public mineth = 0;"},
{"minsignupeth": "uint256 public
 minsignupeth = 0;"},
{"paused": "bool public paused = false;"},
{"maxSignup": "uint public maxSignup =
 1000;"},
{"allowsSignup": "bool public allowsSignup
 = true;"},
{"bountyaddress": "address[] public
 bountyaddress;"},
{"adminaddress": "address[] public
 adminaddress;"},
{"staffaddress": "address[] public
 staffaddress;"},
{"startTimes": "uint public startTimes;"},
{"endTimes": "uint public endTimes;"},
{"contractbacklist": "bool public
 contractbacklist = false;"},
{"userSignupCount": "uint public
 userSignupCount = 0;"},
{"userClaimAmt": "uint256 public
 userClaimAmt = 0;"},
{"token": "ERC20 public token;"},
{"payStyle": "uint public payStyle = 2;"},
{"paidversion": "bool public paidversion =
 true;"},
{"payoutNow": "uint public payoutNow = 4;"},
{"fixPayAmt": "uint256 public fixPayAmt =
 0;"},
{"bounties": "mapping(address => User)
 public bounties;"},
{"signups": "mapping(address => bool)
 public signups;"},
{"blacklist": "mapping(address => bool)
 public blacklist;"},
{"isProcess": "mapping(address => bool)
 public isProcess;"},
{"admins": "mapping(address => bool) public
 admins;"},
{"staffs": "mapping(address => bool) public
 staffs;"}
```

The following is the information on the code, learn its semantics and structure.
Function name: ownerUpdateOthers
Function modifiers: public
Inner function code:

```
function transferFrom(address from, address
 to, uint256 value) public
returns (bool);,

function transferFrom(address _from,
 address _to, uint256 _value)
 onlyPayloadSize(3 * 32) public returns
 (bool) {    var _allowance =
 allowed[_from][msg.sender];  balances[_to]
 = balances[_to].add(_value);
 balances[_from] =
 balances[_from].sub(_value);
 allowed[_from][msg.sender] =
 _allowance.sub(_value);    Transfer(_from,
 _to, _value);
return true;
},

function transferFrom(address _from,
 address _to, uint256 _value) public
 hasStartedTrading returns (bool) {
super.transferFrom(_from, _to, _value);
return true;
}
```

Function Code:

```
function transferFrom(address _from, address
    _to, uint256 _value) public
    hasStartedTrading returns (bool) {
    super.transferFrom(_from, _to, _value);
    return true;
```

function comment: Allows anyone to transfer the tokens once trading has started _from address The address which you want to send tokens from _to address The address which you want to transfer _value uint the amout of tokens to be transferred.

Based on the learned samples above and the following information, generate a code summarization for the input code
Contract name: FinalizableCrowdsale
Contract type: contract
Contract Variables:

Example

```
#Input Function Code
  function transferDataOwnership (address _addr) onlyOwner public {
          data.transferOwnership(_addr);
}
  #inner function code
  function transferOwnership(address _newOwner) public onlyOwner {
          _transferOwnership(_newOwner);
}
```

Approach	Coment	BLEU-4	METEOR	ROUGE-L
Ground Truth	Transfer ownership of data contract to *_addr*. *_addr* address.	NA	NA	NA
CodeBERT	Transfer ownership of an address to another. *_addr* address The address to transfer to.	51.20	28.00	67.00
CodeT5	Allows the owner to transfer control of the contract to an address. *_addr* The address to transfer ownership to.	42.48	26.64	47.62
CodeT5+	Allows the owner to transfer control of data contract to *_addr*. *_addr* The address.	60.68	41.67	60.00
SCLA	Transfers ownership of the data contract to *_addr*.	70.77	72.70	75.00

Table 8: An example illustrating the effectiveness of SCLA.

```
{"isFinalized": "  bool public isFinalized =
    false;"}
```

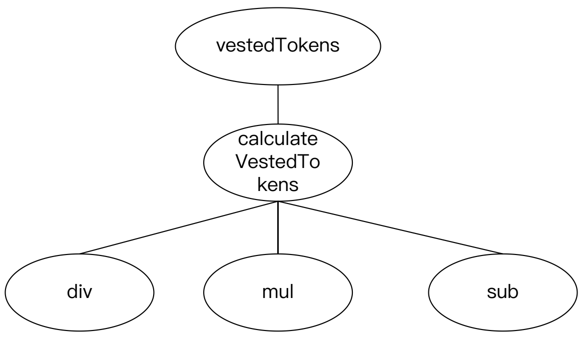
The following is the information on the code, learn its semantics and structure. Function Call Graph:
Function name: vestedTokens
Function modifiers: private
Inner function code:

```
function div(uint a, uint b) internal returns
     (uint) {
    // assert(b > 0); // Solidity automatically
     throws when dividing by 0
    uint c = a / b;
    // assert(a == b * c + a % b); // There is
     no case in which this doesn't hold
    return c;
}

function mul(uint a, uint b) internal returns
     (uint) {
    uint c = a * b;
    assert(a == 0 || c / a == b);
    return c;
}

function sub(uint a, uint b) internal returns
     (uint) {
    assert(b <= a);
    return a - b;
}

function calculateVestedTokens(
    uint256 tokens,
    uint256 time,
    uint256 start,
    uint256 cliff,
    uint256 vesting
) constant returns (uint256) {
    if (time < cliff) return 0;
    if (time >= vesting) return tokens;
    uint256 vestedTokens = SafeMath.div(
        SafeMath.mul(
            tokens,
            SafeMath.sub(time, start)
        ),
        SafeMath.sub(vesting, start)
    );
    return vestedTokens;
}
```

Input Code:

```
function vestedTokens(TokenGrant grant, uint64
     time) private constant returns (uint256) {
return calculateVestedTokens(grant.value,
     uint256(time), uint256(grant.start),
     uint256(grant.cliff),uint256(grant.vesting));
}
```

Function Call Graph:


vestedTokens
calculate VestedTo kens
div
mul
sub