CFStra: Enhancing Configurable Program Analysis Through LLM-Driven Strategy Selection Based on Code Features

Jie Su[1], Liansai Deng[1], Cheng Wen[1], Shengchao Qin[1,2(✉)], and Cong Tian[1,2(✉)]

[1] Guangzhou Institute of Technology, Xidian University, Xi'an, China
{sujie01,wencheng}@xidian.edu.cn, 23031212358@stu.xidian.edu.cn, shengchao.qin@gmail.com, ctian@mail.xidian.edu.cn
[2] ICTT and ISN Laboratory, Xidian University, Xi'an, China

Abstract. Configurable Program Analysis (CPA) allows users to customize program analysis based on their preferences. However, current program verification tools like CPAchecker require manual strategy selection, which can be complex and error-prone. In this paper, we present a novel approach to efficiently perform program verification tasks by harnessing the capabilities of Large Language Models (LLMs) to automatically select verification strategies based on code features and specifications. Specifically, we begin by extracting relevant code snippets and querying LLMs to identify code features. Based on the identified code features, we propose a strategy selector to automatically choose the verification strategy. Finally, we execute the CPAchecker with the selected verification strategy. We evaluated our approach using a diverse set of 600 verification tasks. The results demonstrate the effectiveness of our approach, surpassing basic strategies and SOTA combination strategies while also standing out for its simplicity and ease of understanding.

Keywords: Program Verification · Strategy Selection · Large Language Model · Code Feature

1 Introduction

Background and Problem. Program verification is a complex and tedious task as it often requires users to specify the desired properties of the program, choose the appropriate verification technique and tool, and provide the necessary strategy or other configuration parameters. Different verification tools, algorithms, abstract domains, and configurations, coexist with their different strengths in terms of approaching a verification problem.

Configurable Program Analysis (CPA) [5,31] is a technique that allows users to customize the analysis of programs based on their specific needs and preferences. The well-known framework CPAchecker simplifies configuration and

W.-N. Chin and Z. Xu (Eds.): TASE 2024, LNCS 14777, pp. 374–391, 2024.
https://doi.org/10.1007/978-3-031-64626-3_22

automates analysis and verification tasks, supporting techniques like predicate abstraction and model checking. CPAchecker also provides a rich set of configuration parameters that can be used to adjust the precision, performance, and resource consumption of the analysis. However, the default configuration often inadequate for large and complex programs, requiring manual selection of advanced verification strategies. This often necessitates a deep understanding of the target source code and the use of expert knowledge. This process can be arduous and time-consuming, especially for inexperienced users.

Existing Work. To address the usability issues of existing verification tools, numerous techniques for verification strategy selection have been proposed. For example, Dirk *et al.* [3] presented a strategy selection method that is based on a straightforward model using a succinct set of Boolean features extracted statically from source code. This approach involves focusing on simple features while excluding more intricate ones such as recursion and concurrency, thus limiting its efficacy in handling complex programs. In addition to this, various techniques utilizing machine learning have been suggested to automatically choose a promising verification strategy, as demonstrated by recent developments [14,15]. One approach involves using implicit features derived from the input program to forecast a strategy ranking, eliminating the need to explicitly define these features [14]. The learner is provided with the program dependency graph, a directed graph capable of illustrating the data, control, and concurrency dependencies within a program, enabling it to deduce the necessary characteristics. Nevertheless, this technique is overly burdensome and constrained in its scalability.

Insight. AI-powered Generative Large Language Models (LLMs) routinely make tremendous progress in software engineering tasks. Examples include program synthesis from natural language descriptions by GPT-4 [33] or Github Copilot [50], solving competitive programming problems with AlphaCode [29], generating loop invariant with ChatGPT [10,35], and generating entire proofs of theorems using fine-tuned LLMs [21], among others. Many of these tasks require strong reasoning skills over code and code understanding. This raises the question of whether we can utilize the reasoning skills of LLMs in the formal setting of software verification and complement existing automatic verifiers such as CPAchecker. Our key insight is that LLMs possess advanced program comprehension capabilities, allowing them to serve as proficient human experts in identifying code features and selecting verification strategies.

Contribution. In this paper, we propose a novel approach to effectively complete the verification task by driving the CPAchecker on selecting verification strategies based on code features, and harnessing (some of) the powers of Large Language Models (LLMs). We begin by extracting relevant code snippets from the target program and querying LLMs to identify their code features. Based on these identified code features, we propose a strategy selector to automatically choose the verification strategy that is most suitable for the program. Finally, we execute the CPAchecker with the configuration corresponding to the selected verification strategy. Our main contributions are:

```
int main() {
    struct node *list = create_list();
    struct node *low = NULL;
    struct node *high = NULL;
    // Split list into low and high
    struct node *p = list;
    while (p) {
        struct node *l = p->value >= 0 ? high : low;
        struct node *next = p->next;
        p->next = l;
        l = p;
        p = next;
    }
    // Check that low and high contain expected elements
    while (low) {
        if (!(low->expected_list == LOW))
            {reach_error();}
        low = low->next;
    }
    ...
}
```

Listing 1.1. An Motivating Example of quick_sort_split.c, from SV-COMP

- *Novelty.* We present a fast, general, and easily extensible approach to efficiently perform program verification tasks by harnessing the capabilities of LLMs to automatically select verification strategies based on code features.
- *Practical Approach.* We identify and address several practical challenges by combining code feature identification via LLMs, prompt engineering, and customized verification strategy selection.
- *Open-source Implementation.* We have developed and implemented our proposed approach as a tool named CFStra. We have made the implementation, along with all relevant publicly available data, accessible to facilitate comparison: https://sites.google.com/view/cfstra/.
- *Evaluation.* We extensively evaluate our approach using a diverse set of 600 verification tasks against 8 basic strategies and 4 SOTA combination strategies (*i.e.*, CPA-Seq, BMC-BAM$_R$-PA, VA-BAM$_R$-KI, PIChecker) to demonstrate its effectiveness, efficiency, and scalability.

2 Background and Motivation

In this section, we motivate our approach by employing a motivating example for SV-COMP, as well as providing the necessary background on LLMs.

Consider the code snippet that implements a quick sort splitting algorithm for a linked list, as shown in Listing 1.1. To verify its correctness, users need to identify the relevant code features like pointers, dynamic memory allocation, and input-dependent loops. Choosing appropriate verification strategies like predicate abstraction or k-induction significantly impacts precision and performance. However, manually identifying and selecting strategies can be challenging due to limited understanding, lack of expert knowledge, and dealing with numerous verification tasks.

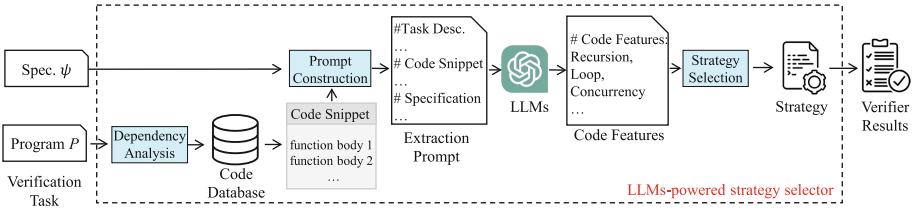

Fig. 1. The workflow of our approach

Therefore, there is a need for an automated and intelligent approach that can leverage the program comprehension capabilities of LLMs to drive the CPAchecker. LLMs are neural network models that are trained on large corpora of natural and programming languages, and can generate natural language texts or code snippets based on given inputs or queries. LLMs have shown remarkable abilities in understanding and generating programs, such as code completion, code repair, and code synthesis. We hypothesize that LLMs can also serve as proficient human experts in identifying code features for selecting verification strategies. By using LLMs, we can reduce the manual effort and the cognitive load of the users, and improve the efficiency and the effectiveness of the program verification.

3 Methodology

In this section, we describe the main components and steps of our approach to automate program verification by driving the CPAchecker based on code features. Figure 1 shows an overview of our approach, which consists of three phases: code snippets extraction, code feature identification, and strategy selection.

3.1 Code Snippet Extraction

When manually analyzing a program, it is common not to expensively analyze the entire source code. Instead, this process typically focuses on some key functions such as the main entry and the relevant functions it invokes. These functions and their intrinsic dependencies often reflect the structure, semantics, and behavioral features of the entire program. Based on this observation, CFStra derives only the key functions enriched with necessary calling contexts. To achieve this, CFStra extracts code snippets of the key functions by performing dependency analysis on the provided source program.

During the process of dependency analysis, the program dependency graph [19], denoted as $G = (N, E)$, is utilized to explicitly represent the dependencies for each operation in the program. It consists of a set of nodes N representing statements and the variables defined in the program, along with a set of directed edges $E \subseteq N \times N$ corresponding to data and control dependencies [1] between the nodes. In addition, we also augment the graph with concurrency-related dependence to extend its applicability: **Data dependence**: Each edge $e \in E$

Algorithm 1: Code Snippet Extraction through traversal on program dependency graph

Input : A program dependency graph $G = (N, E)$, and an initial key function f.
Output: A code snippet S including a set of key function bodies.

```
1  begin
2      // Search the graph node corresponding to the given function.
3      n := search_node(G, f);
4      W ← {n}, I ← ∅;
5      // Traverse the dependency graph to collect all the related key functions.
6      while W ≠ ∅ do
7          n := pop(W);
8          if ¬visited(n) then
9              // Get all the nodes that have data/control/concurrency dependence
                with n in G.
10             W := W ∪ {n′|(n, n′) ∈ E};
11             I := I ∪ {n};
12     // Collect all the key functions.
13     S ← ∅;
14     for n ∈ I do
15         // Retrieve the function containing this node.
16         s := retrieve_function(n);
17         if s ∉ S then
18             S := S ∪ {s};
19     return S;
```

is directed from one statement to the other statements that reference the same variable; **Control dependence**: Each edge $e \in E$ is directed from one statement to an if-statements; **Concurrency dependence**: Each edge $e \in E$ directed from: 1. a thread creation statement (e.g., pthread_create) to the entry node of the thread template function; 2. the return statement of thread template function to the statement (e.g., pthread_join) waiting for the thread to finish.

Algorithm 1 shows the details of extracting the code snippets. It takes two inputs: an initial key function f and a constructed program dependency graph G. The initial key function can be the entry function of the entire program or be determined based on error information such as program exception stacks and defect reports. In our algorithm, we define the extracted code snippet as S, containing key function bodies associated with input function f. To extract S, we initiate from the initial entry function node and traverse connected nodes in G using depth-first search (DFS) manner (lines 6–11) until all reachable nodes are gathered in set I (line 11). Subsequently, functions corresponding to nodes in I will be retrieved and stored in set S (lines 14–18), representing the key functions invoked or indirectly related to the initial key function f. The extracted code snippet S provides the necessary calling context and information for language models to analyze code features and mitigates the existing constraint of mainstream language models in handling lengthy texts.

Prompt

I am an expert in code analysis and verification, possessing a profound ability to comprehend and analyze program source code.

Task Description
The code snippet will be provided to me for the purpose of extracting the code features (e.g., loop, recursion, concurrency, array, pointer deference, etc.). The code features should be concluded in the format of 'Code Features: F1, F2, ...', where F1, F2 are specific program structure, complex data type, or operations.

Code Snippet
```C
<CODE_SNIPPET>
```

Fig. 2. The simplified prompt template we used

In the above algorithm, although the utilization of precise program dependency graphs can effectively improve the completeness of extracted key functions, creating precise dependency relationships for a large-scale program is typically time-consuming and not scalable. Considering the strong correlation between dependency relationships and intrinsic syntactic structure in source code, we devise a lightweight and efficient approach to improve the practicality of the code snippet extraction step. Specifically, CodeQuery [37], a pattern-based static analysis approach, is employed to construct a comprehensive code database, allowing for the retrieval of symbol references, global definitions, callee and caller functions. This enables us to efficiently analyze dependency relationships by leveraging the retrieved information.

3.2 Code Feature Identification

The second phase of our approach involves identifying the code features within the target program's source code utilizing LLMs. Code features are the characteristics or properties of the code that affect the choice of verification strategy, such as arrays, pointers, floating points, loops, concurrency, recursion, *etc.* Code features can be syntactic, semantic, or behavioral, depending on the level of abstraction and analysis.

To identify code features, we use LLMs that are trained on large corpora of natural and programming languages, such as ChatGPT [47]. LLMs have shown remarkable abilities in understanding and generating programs, such as code completion, code synthesis, and code repair. We hypothesize that LLMs can also serve as proficient human experts in identifying code features, by using their natural language and program comprehension capabilities.

To use LLMs, we formulate the code feature identification task as a natural language query and response problem. For the extracted code snippet, we generate a natural language query that asks the LLM to list the code features of the snippet. The prompts mainly comprise three key elements: role declaration, task description, and code snippet. Specifically, CFStra creates a document with contents in Markdown format to construct the prompt. As shown in Fig. 2, the

Response

- **Pointers**: The code extensively uses pointers to manipulate linked list nodes.
- **Struct Definition**: The code defines and uses a structure (struct node) to represent nodes in a linked list.
- **Loop**: The code contains two while loops for iterating through the linked list.
- **Conditional Statements**: Conditional statements are used to determine whether a node should be added to the low or high list based on the value of p->value.
- **Function Calls**: The code calls functions create_list() and reach_error().

Code Features: pointer operations, struct data type, loop, conditional statements, function calls.

Fig. 3. The response from LLMs

role declaration statement is provided in the first line, while the instructions are placed below the "Task Description" section. Besides, the extracted code snippets are presented in blue text. Furthermore, the "Task Description" section concludes with a sentence that instructs the desired output format. This output format is critical as it requires programmatic processing of the LLM's response. We then feed the query and the code snippet to the LLM, and expect the LLM to generate a natural language response that lists the code features of the snippet. For example, the response for extracting code features of the quick sort splitting algorithm is shown in Fig. 3.

We then parse the response and extract the code features from it, and store them in the database along with the code snippet. In order to balance the diversity and randomness of code features extracted from the source code, avoid the problem of insufficient consideration of the core features that affect strategy selection due to the high randomness of the output, and the problem of inadequate code feature extraction caused by low randomness output. We sample the query response multiple times by using the same prompt while applying a relatively high temperature for LLM to capture more comprehensive code features. Thereafter, we count and retain the extracted features that appear in more than half of the query instances. Intuitively, the retained code features appear frequently in the query results and are highly representative.

3.3 Strategy Selection

The third phase of our approach is to select the verification strategy for the program to be verified, based on the identified code features. A verification strategy is a combination of verification techniques and configuration parameters that are used to analyze and verify the target program.

The description of our four verification strategies will refer to the components: **VA-NoCEGAR**: value analysis without CEGAR [7]; **VA-CEGAR**: value analysis with CEGAR [7]; **PA**: predicate analysis with CEGAR [6]; **KI**: k-induction with continuously refined invariant generation [4]; $\mathbf{BAM_R}$: block-abstraction memorization (BAM) for a composite abstract domain of predicate analysis and value analysis [45]; **BMC**: bounded model checking (BMC) [9]; **C-Intp**: conditional interpolation empowered predicate analysis with CEGAR [39]; **PC-DPOR**: prioritized constraint-aided partial-order reduction [40]. The first

six components are primarily used for verifying sequential programs, while the last two components are mainly employed for verifying concurrent programs.

The set of four verification strategies that we use in our strategy classifier are based on components from the above list: **CPA-Seq** sequentially combines VA-NoCEGAR, VA-CEGAR, PA, KI, and BAM_R. Any of the components may terminate early if it detects that it cannot handle the task. If none of the four components VA-NoCEGAR, VA-CEGAR, PA and KI can handle the task, and if KI fails due to the task requires handling of recursion, the BAM_R component runs. If either VA-NoCEGAR or VA-CEGAR find a bug, the feasibility of error path will be checked; if the check passes, the bug is reported, otherwise, the component result is disregarded and the subsequent component is executed. **BMC-BAM_R-PA** is a sequential combination of BMC, BAM_R, and PA. As above, any of the components may terminate early if it detects that it cannot handle the task. If the first component BMC fails because the task requires handling of recursion, the BAM_R component runs; if the reason why BMC fails was not recursion or if BAM_R fails to solve the task, PA runs. In this strategy, BAM_R and PA are only used as fallback components if the BMC components fails due to recursion or other unsupported features. **VA-BAM_R-KI** sequentially combines VA-NoCEGAR, VA-CEGAR, BAM_R, and KI. Any of the components may terminate early if it detects that it cannot handle the task. As in CPA-Seq, if either VA-NoCEGAR or VA-CEGAR find a bug, the feasibility of error path will be checked; if the check passes, the bug is reported. If the reason why VA-CEGAR failed was not recursion or if BAM_R also fails to solve the task, KI runs. **PIChecker** [41] is a sequential combination of PC-DPOR, C-Intp (MathSAT5 back-end [12]), C-Intp (SMTInterpol back-end [11]). If a counterexample is reported by PC-DPOR, the feasibility of this error path will be checked. If PC-DPOR fails due to some unsupported features such as array operations, the verification will continue by using the other two C-Intp based components with different back-end solvers.

To determine the verification strategy of a program, we employ a strategy classification based on the extracted code features. The strategy classifier is a function comprises the four verification strategies mentioned above:

$$\mathit{Strategy} = \begin{cases} \text{PIChecker}, & \text{if} \quad \text{hasConcurrency} \\ \text{BMC-BAM}_\text{R}\text{-PA}, & \text{if} \quad \text{not hasLoop} \\ \text{VA-BAM}_\text{R}\text{-KI}, & \text{if} \quad \text{hasLoop} \wedge \text{hasComplexType} \\ \text{CPA-Seq}, & \text{Otherwise} \end{cases} \tag{1}$$

It is defined to always choose the strategy PIChecker if 'Concurrency' is present in the extracted code features, since PIChecker is the only composite strategy used for verifying concurrent programs. If 'Loop' is not in the code features, the strategy BMC-BAM_R-PA will be chosen, as there is no need to perform any potentially expensive invariant-generating algorithm if no loop exists in the program. If 'Loop' is in the code features and there are complex data types (e.g., struct, union, array, and floating point type), it chooses VA-BAM_R-KI. If 'Loop' is in the code features and there are no complex data types, it chooses the CPA-Seq strategy.

4 Evaluation

We have developed a prototype tool called CFStra and conducted a comprehensive evaluation to evaluate its effectiveness. This study aims to compare CFStra with various fixed strategies, including basic strategies and sequential combination strategies. This study also highlights the implications of each proposed improvement through an ablation study.

Table 1. Detail information of Benchmark

No.	Dataset	Files	#KLOC
1	ReachSafety	366	1668.2
2	MemSafety	164	236.0
3	ConcurrencySafety	70	55.7
Total		600	1959.9

4.1 Experimental Setup

Benchmark Set. The set of verification tasks that we use in our experiments is taken from the benchmark collection that is also used in SV-COMP. We use all verification tasks from the benchmark collection for which we have identified different strategies. We conducted the experiments on the 600 verification tasks in the *ReachSafety*, *MemSafety*, *ConcurrencySafety* from SV-COMP 2023. The detailed information is presented in Table 1.

Comparation Among Existing Strategies. We first selected 8 widely used basic strategies to demonstrate their diverse performance. Subsequently, we compared CFStra with 4 existing SOTA sequential combination strategies (*i.e.*, CPA-Seq, BMC-BAM$_{\mathrm{R}}$-PA, VA-BAM$_{\mathrm{R}}$-KI, and PIChecker) for comparison, which incorporate various basic verification strategies. In the ablation study, we also include two simplified versions of CFStra.

Performance Metrics. We evaluate the performance of all competitors based on the verification correctness rate, time efficiency, and memory efficiency.

Configuration of LLMs. All interactions with LLMs, including sending requests and receiving responses, are conducted through ChatGPT's API. We specifically employ version *gpt-3.5-turbo* of ChatGPT. Various hyperparameters are involved in utilizing the APIs offered by ChatGPT [25]. For our configuration, we have set the values of `max_token` to 2048, `temperature` to 0.7, and query times to 7.

Experiment Infrastructure. All our experiments were conducted by version 2.3 of CPAchecker on machines equipped with 2.2GHz CPU(AMD EPYC 7773x), featuring 128 processing units and 503.7 GB of RAM, using OpenJDK 17 and

Table 2. Results for all 600 verification tasks, for 8 widely used basic strategies

Approach	VA-NoCEGAR	VA-CEGAR	PA	KI	BAM_R	BMC	CIntp	PC-DPOR
Score	75	53	143	255	−452	40	−685	58
Correct results	67	49	103	201	41	35	43	81
Correct proofs	24	20	56	150	19	21	8	41
Correct alarms	43	29	47	51	22	14	35	40
Wrong proofs	0	0	0	2	0	0	1	1
Wrong alarms	1	1	1	2	32	1	44	2
Timeouts	163	106	107	125	168	21	308	208
Out of memory	25	14	1	11	2	1	1	1
Other inconlusive	344	430	388	261	357	542	204	258
Times for correct results								
Total CPU Time(h)	1.5	1	1	1.3	1.2	0.5	2.2	2.3
Avg. CPU Time(s)	9	6	6	7.8	7.2	3	13	14
Total Wall Time(h)	1.5	1	1	1.3	1.2	0.5	2.2	2.3
Avg. Wall Time(s)	9	6	6	7.8	7.2	3	13	14

running on a 64-bit Ubuntu LTS 20.04.5. Each verification run was restricted to two CPU cores, with a maximum runtime of 15 min and a memory allocation of 15 GB. The benchmarking framework BenchExec [8] was employed to oversee the experiments, ensuring precise and dependable measurements.

4.2 Performance of Different Strategies

Table 2 shows the results of applying 8 different basic strategies to 600 verification tasks. These verification strategies are: VA-NoCEGAR, VA-CEGAR, PA, KI, BAM_R, BMC, C-Intp, and PC-DPOR. The table assesses each strategy's performance based on score, correctness, time, and memory.

The results from Table 2 demonstrate that the basic verification strategies yield different outcomes for the same verification task. For instance, KI exhibits the highest score and the highest number of correct proofs, but also the highest number of incorrect proofs. On the other hand, BMC has the fewest timeouts and out of memory, but also the lowest number of correct results. The high frequency of inaccurate outcomes produced by BAM_R and C-Intp, along with the consequent low scores, can be attributed to the absence of support for pointer alias processing. Hence, selecting the appropriate strategy is crucial to strike a balance between accuracy, efficiency, and comprehensiveness.

Figure 4 displays the quantile functions for the 8 basic strategies examined. It is evident that KI significantly outperforms the other strategies in these experiments. However, the performance of KI based combination strategy (i.e., VA-BAM_R-KI) still lags behind our proposed approach, CFStra, and other sequential combination strategies, as depicted in Fig. 5. This is why our approach focuses on selecting existing combination strategies only, as demonstrated in Formula 1. The subsequent section will delve into a detailed analysis of the results obtained with CFStra and compare them with other sequential combination strategies.

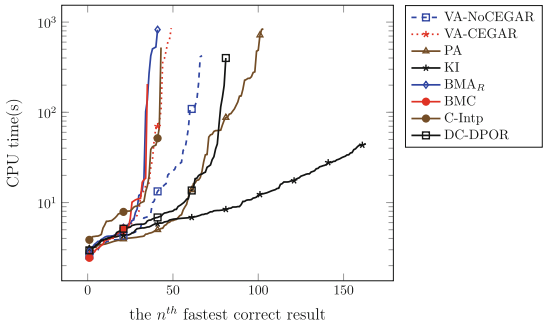

Fig. 4. The quantile plot of time consumption in logarithmic scale

Table 3. Results for all 600 verification tasks for combination strategies

Approach	CFStra	PIChecker	CPA-Seq	BMC-BAM$_R$-PA	VA-BAM$_R$-KI
Score	469	278	225	111	224
Correct results	315	228	144	103	144
Correct proofs	202	146	97	40	96
Correct alarms	113	82	47	63	48
Wrong proofs	0	1	0	0	0
Wrong alarms	3	4	1	2	1
Timeouts	224	319	187	234	186
Out of memory	13	2	24	8	25
Other inconclusive	45	47	244	253	244
Times for correct results					
Total CPU Time(h)	1.4	2.3	1.5	2	1.5
Avg. CPU Time(s)	8.4	14	9	12	9
Total Wall Time(h)	2.2	2	1.5	2	1.5
Avg. Wall Time(s)	13	14	9	12	9

4.3 Compared to Other SOTA Combination Strategies

Table 3 displays the results of CFStra, comparing it with four SOTA combination strategies (*i.e.*, PIChecker, CPA-Seq, BMC-BAM$_R$-PA, VA-BAM$_R$-KI). Even though the sequential combination strategy significantly outperformed individual basic strategies, it remains a fixed approach. CFStra achieved notably higher scores than each of the other combination strategies. The adoption of the LLM-driven strategy proved to be advantageous, resulting in a superior score compared to competitors. While this method may lead to more false alarms than other strategies, it enhances the accuracy of results significantly, effectively resolving this issue. These findings suggest that utilizing the strategy selector based on the code features identified by LLMs can lead to favorable outcomes, even when choices are limited. Furthermore, CFStra exhibits enhanced efficiency in task identification, with only 45 inconclusive tasks out of 600. The results indicate that the use of LLM-driven strategy selection is more effective than consistently selecting a fixed strategy. As a result, the strategy selector employed in *CFStra* strikes a balance between accuracy, efficiency, and comprehensiveness.

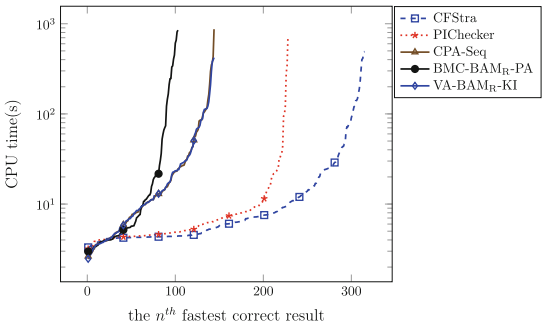

Fig. 5. The quantile plot of CFStra, compared with 4 combination strategies.

Table 4. Results for all 600 verification tasks for 3 version of CFStra

Approach	CFStra$_{direct}$	CFStra$_{llm}$	CFStra
Score	224	229	469
Correct results	143	147	315
Correct proofs	97	98	202
Correct alarms	46	49	113
Wrong proofs	0	0	0
Wrong alarms	1	1	3
Timeouts	180	183	224
Out of memory	24	25	13
Other inconlusive	252	244	45
Times for correct results			
Total CPU Time(h)	2.3	1.2	1.4
Avg. CPU Time(s)	14	7.2	8.4
Total Wall Time(h)	2.5	1.5	2.2
Avg. Wall Time(s)	15	9	13

Figure 5 depicts the quantile functions of CFStra in comparison with four combination strategies. It is clearly evident from the curves in the plot that PIChecker outperforms BMC-BAM$_R$-PA and VA-BAM$_R$-KI significantly, but still trails behind CFStra. Upon comparing the experimental results, it was discovered that within the 95 tasks in the *ReachSafety* dataset, PIChecker failed to achieve effective verification outcomes due to verification timeouts and other reasons. Conversely, CFStra selected more appropriate strategies based on the extracted code features, making these tasks can be effectively verified with limited resources. Therefore, the superiority of CFStra over the other combination strategies is clearly evident.

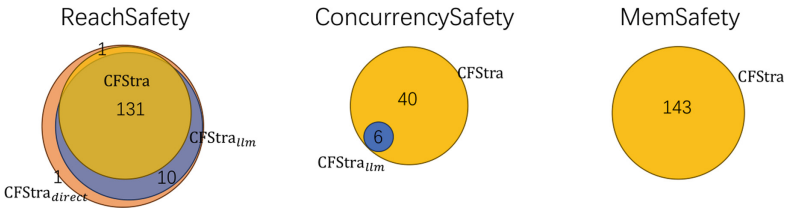

Fig. 6. The number of verification tasks successfully verified by CFStra$_{direct}$, CFStra$_{llm}$ and CFStra across different categories.

4.4 Ablation Study

To evaluate the effectiveness of our proposed approach, we performed an ablation study to compare the performance of CFStra with its two simplified versions: CFStra$_{direct}$ and CFStra$_{llm}$. CFStra$_{direct}$ directly utilizes LLM for strategy selection, without analyzing code features. CFStra$_{llm}$ decomposes the strategy selection process into two steps: first, analyzing code features using LLM, and then predicting the strategy using LLM. We used the same set of 600 verification tasks as in the previous section, where the results are shown in Table 4. We can observe that CFStra significantly outperforms both CFStra$_{direct}$ and CFStra$_{llm}$. This indicates that CFStra can effectively select the most appropriate verification strategy for each verification task, based on the code features and specifications. Moreover, CFStra achieves comparable or better CPU and wall time than CFStra$_{direct}$, demonstrating that CFStra is efficient and scalable.

Figure 6 presents the comparison of the success of the verification task among three versions of CFStra in various categories in Table 1: ReachSafety, ConcurrencySafety. The results indicate that CFStra consistently outperforms the other versions across all categories, particularly excelling in ConcurrencySafety and MemSafety by successfully verifying a notable number of tasks. This demonstrates that CFStra can effectively choose the appropriative strategy to handle complex and challenging verification tasks that involve concurrency and memory issues. On the other hand, CFStra$_{llm}$ slightly outperforms CFStra$_{direct}$ in verifying tasks within the ConcurrencySafety category, attributed to the decomposed strategy selection process into two steps. Nevertheless, the overall performance between CFStra$_{direct}$ and CFStra$_{llm}$ remains comparable. This suggests a need for integrating code features and specifications into the independent strategy selection process, as CFStra does in the formula 1.

The ablation study also reveals the limitations of CFStra$_{direct}$ and CFStra$_{llm}$. CFStra$_{direct}$ suffers from low accuracy and high inconclusiveness, as it does not take into account the code features that are crucial for strategy selection. CFStra$_{llm}$ improves slightly over CFStra$_{direct}$, as it analyzes code features using LLM, but it still fails to select the best strategy for many verification tasks, as it relies on LLM for the final prediction. This suggests that LLM alone is not sufficient for strategy selection, and that a more sophisticated strategy

selector is needed. In summary, the ablation study confirms the superiority of CFStra over its two simplified versions, and validates the design choices of our approach.

4.5 Threats to Validity

There are three major validity threats. First, One potential threat is the selection bias that may arise from using only 600 verification tasks from SV-COMP, which could limit the diversity of our dataset. To address this concern, we made sure to select diverse programs from various domains with different characteristics. Second, the generalizability of different LLMs is also a concern. We have also implemented CFStrato a popular and open-source LLM called *Llama-2*. Similar results were observed in our preliminary experiments. Third, due to their inherent randomness, LLMs generate varied answers for identical programs to be verified with the same prompts. A potential threat to the validity of our study is that conclusions drawn from random results may be misleading. To address the above three potential threads, future works involve conducting multiple experiments on a substantial dataset.

5 Related Work

In this section, we review the existing literature on program verification, as well as LLMs for program analysis and verification, and explain how our work is different from or builds upon them.

Program Verification and its Strategies. Program verification is a well-studied and active research area, that aims to ensure the correctness and reliability of software systems [20,38]. There are various techniques and tools for program verification, such as static analysis [31,49], model checking [13,26], theorem proving [32,34], runtime verification [2,16], *etc.* Several studies have sought to combine various verification strategies sequentially, (*e.g.,* CPA-Seq and PeSCo) [31,36], which prove to be more effective than each strategy used alone. However, these methods are either limited in scope or scalability, or rely on external sources. Moreover, these methods do not exploit the program comprehension capabilities of LLMs, which can serve as proficient human experts in identifying code features and selecting verification strategies.

LLMs for Program Analysis and Verification. LLMs are neural network models that are trained on large corpora of natural and programming languages [30,51]. LLMs have demonstrated impressive capabilities in comprehending and producing code, leading to a surge in interest in utilizing LLMs for aiding in program analysis and verification tasks [17,22]. These tasks include static analysis [28,43], program testing [27,42], program verification [44,46], bug reproduction [23,24] and bug repair [18,48]. Recent advancements in LLMs for verification have led to the development of Baldur [21], a proof-synthesis tool

that utilizes transformer-based pre-trained LLMs, fine-tuned proofs, to generate and repair complete proofs. In contrast, our approach centers on leveraging LLMs to detect code features and guide the verification tool in selecting strategies, whereas Baldur primarily automates the generation of proofs for theorems. Moreover, to the best of our knowledge, there is no comparable existing work that tries to drive program verification by using LLMs to identify code features and select verification strategies, which are the key steps of our approach.

6 Conclusions

This paper presents a novel approach for automating program verification by selecting verification strategies based on code features. Our approach leverages the program comprehension abilities of Large Language Models (LLMs) to recognize code features. Subsequently, a selection model is then developed to automatically choose appropriate verification strategies. We evaluated the effectiveness and efficiency of our approach using a substantial collection of verification tasks from SV-COMP. The outcomes indicate that our approach not only surpasses basic strategies and SOTA combination strategies but also stands out for its simplicity and ease of understanding.

Acknowledgements. The authors would like to thank the anonymous reviewers for their constructive comments. This work was supported in part by the National Natural Science Foundation of China (Nos. 62302375, 62192734, 62193273024), the China Postdoctoral Science Foundation funded project (No. 2023M723736), and the Fundamental Research Funds for the Central Universities.

References

1. Allen, J.R., Kennedy, K., Porterfield, C., Warren, J.: Conversion of control dependence to data dependence. In: Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, pp. 177–189 (1983)
2. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Lectures on Runtime Verification: Introductory and Advanced Topics, pp. 1–33 (2018)
3. Beyer, D., Dangl, M.: Strategy selection for software verification based on boolean features. In: Margaria, T., Steffen, B. (eds.) ISoLA 2018. LNCS, vol. 11245, pp. 144–159. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03421-4_11
4. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 622–640. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_42
5. Beyer, D., Keremoglu, M.E.: CPAchecker: a tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_16
6. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. In: Formal Methods in Computer Aided Design, pp. 189–197. IEEE (2010)

7. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37057-1_11
8. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transfer **21**, 1–29 (2019)
9. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. In: Handbook of Satisfiability, vol. 185, no. 99, pp. 457–481 (2009)
10. Chakraborty, S., et al.: Ranking LLM-generated loop invariants for program verification. arXiv preprint arXiv:2310.09342 (2023)
11. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: an interpolating SMT solver. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 248–254. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31759-0_19
12. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The MathSAT5 SMT solver. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 93–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_7
13. Clarke, E.M.: Model checking. In: Ramesh, S., Sivakumar, G. (eds.) FSTTCS 1997. LNCS, vol. 1346, pp. 54–56. Springer, Heidelberg (1997). https://doi.org/10.1007/BFb0058022
14. Czech, M., Hüllermeier, E., Jakobs, M.C., Wehrheim, H.: Predicting rankings of software verification tools. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Software Analytics, pp. 23–26 (2017)
15. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Formal Methods Syst. Des. **50**, 289–316 (2017)
16. Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. Eng. Dependable Softw. Syst. 141–175 (2013)
17. Fan, A., et al.: Large language models for software engineering: survey and open problems. arXiv preprint arXiv:2310.03533 (2023)
18. Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated repair of programs from large language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 14–20 May 2023, pp. 1469–1481. IEEE (2023)
19. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. **9**(3), 319–349 (1987). https://doi.org/10.1145/24039.24041
20. Fetzer, J.H.: Program verification: the very idea. Commun. ACM **31**(9), 1048–1063 (1988)
21. First, E., Rabe, M., Ringer, T., Brun, Y.: Baldur: whole-proof generation and repair with large language models. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1229–1241 (2023)
22. Hou, X., et al.: Large language models for software engineering: a systematic literature review. arXiv preprint arXiv:2308.10620 (2023)
23. Kang, S., Yoon, J., Askarbekkyzy, N., Yoo, S.: Evaluating diverse large language models for automatic and general bug reproduction. arXiv preprint arXiv:2311.04532 (2023)
24. Kang, S., Yoon, J., Yoo, S.: Large language models are few-shot testers: exploring LLM-based general bug reproduction. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 14–20 May 2023, pp. 2312–2323. IEEE (2023)

25. Kocoń, J., et al.: Chatgpt: jack of all trades, master of none. Inf. Fusion 101861 (2023)
26. Kroening, D., Tautschnig, M.: CBMC – C bounded model checker. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 389–391. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_26
27. Lemieux, C., Inala, J.P., Lahiri, S.K., Sen, S.: Codamosa: escaping coverage plateaus in test generation with pre-trained large language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 14–20 May 2023, pp. 919–931. IEEE (2023)
28. Li, H., Hao, Y., Zhai, Y., Qian, Z.: Assisting static analysis with large language models: a chatgpt experiment. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 2107–2111 (2023)
29. Li, Y., et al.: Competition-level code generation with alphacode. Science **378**(6624), 1092–1097 (2022)
30. Liu, Y., et al.: Summary of chatgpt-related research and perspective towards the future of large language models. Meta-Radiol. 100017 (2023)
31. Löwe, S., Mandrykin, M., Wendler, P.: CPAchecker with sequential combination of explicit-value analyses and predicate analyses. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014. LNCS, vol. 8413, pp. 392–394. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54862-8_27
32. Matthews, J., Moore, J.S., Ray, S., Vroon, D.: Verification condition generation via theorem proving. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 362–376. Springer, Heidelberg (2006). https://doi.org/10.1007/11916277_25
33. OpenAI: GPT-4 technical report. arxiv:2303.08774. View in Article, vol. 2, p. 13 (2023)
34. Ouimet, M., Lundqvist, K.: Formal software verification: model checking and theorem proving. Embedded Systems Laboratory Technical Report ESL-TIK-00214, Cambridge USA (2007)
35. Pei, K., Bieber, D., Shi, K., Sutton, C., Yin, P.: Can large language models reason about program invariants? In: International Conference on Machine Learning, pp. 27496–27520. PMLR (2023)
36. Richter, C., Wehrheim, H.: PeSCo: predicting sequential combinations of verifiers. In: Beyer, D., Huisman, M., Kordon, F., Steffen, B. (eds.) TACAS 2019. LNCS, vol. 11429, pp. 229–233. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-17502-3_19
37. Ruben: A code-understanding, code-browsing or code-search tool. This is a tool to index, then query or search C, C++, java, python, ruby, go and javascript source code. https://github.com/ruben2020/codequery. Accessed 04 Mar 2024
38. Sieber, K.: The Foundations of Program Verification. Springer, Wiesbaden (2013). https://doi.org/10.1007/978-3-322-96753-4
39. Su, J., Tian, C., Duan, Z.: Conditional interpolation: making concurrent program verification more effective. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 144–154 (2021)
40. Su, J., Tian, C., Yang, Z., Yang, J., Yu, B., Duan, Z.: Prioritized constraint-aided dynamic partial-order reduction. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, pp. 1–13 (2022)

41. Su, J., Yang, Z., Xing, H., Yang, J., Tian, C., Duan, Z.: PIChecker: a POR and interpolation based verifier for concurrent programs (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) TACAS 2023. LNCS, vol. 13994, pp. 571–576. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-30820-8_38
42. Tsigkanos, C., Rani, P., Müller, S., Kehrer, T.: Large language models: the next frontier for variable discovery within metamorphic testing? In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 678–682. IEEE (2023)
43. Wen, C., et al.: Automatically inspecting thousands of static bug warnings with large language model: How far are we? ACM Trans. Knowl. Discov. Data (2024)
44. Wen, C., et al.: Enchanting program specification synthesis by large language models using static analysis and program verification. In: International Conference on Computer Aided Verification. Springer, Cham (2024)
45. Wonisch, D., Wehrheim, H.: Predicate analysis with block-abstraction memoization. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 332–347. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-34281-3_24
46. Wu, H., Barrett, C., Narodytska, N.: Lemur: integrating large language models in automated program verification. arXiv preprint arXiv:2310.04870 (2023)
47. Wu, T., et al.: A brief overview of ChatGPT: the history, status quo and potential future development. IEEE/CAA J. Automatica Sinica **10**(5), 1122–1136 (2023)
48. Xia, C.S., Wei, Y., Zhang, L.: Automated program repair in the era of large pre-trained language models. In: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, 14–20 May 2023, pp. 1482–1494. IEEE (2023)
49. Xu, Z., Wen, C., Qin, S.: State-taint analysis for detecting resource bugs. Sci. Comput. Program. **162**, 93–109 (2018)
50. Yetistiren, B., Ozsoy, I., Tuzun, E.: Assessing the quality of github copilot's code generation. In: Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering, pp. 62–71 (2022)
51. Zhao, W.X., et al.: A survey of large language models. arXiv preprint arXiv:2303.18223 (2023)