Записки научных
семинаров ПОМИ
Том 540, 2024 г.

D. Shaikhelislamov, M. Drobyshevskiy, A. Belevantsev

ENSURING TRUSTWORTHY CODE: LEVERAGING A STATIC ANALYZER TO IDENTIFY AND MITIGATE DEFECTS IN GENERATED CODE

Abstract. The rise of large language models (LLMs) has greatly advanced code generation capabilities. A recent StackOverflow survey found that 70% of developers are using or planning to use AI coding tools this year. However, most current methods focus on supervised fine-tuning objectives derived from text generation, often overlooking the distinct sequence-level properties of code, such as compilability, and syntactic and functional correctness. To address this gap, we introduce a novel approach that combines pre-trained LLMs with software analysis tools commonly used to detect vulnerabilities and validate code. Our method leverages detailed feedback from code compilation and analysis, incorporating this specialized knowledge into the prompt chaining process. We present CodePatchLLM, an extension of LLMs that uses Svace feedback for improved code generation. CodePatchLLM is a model-agnostic framework that supports multiple programming languages. Extensive experiments on the LeetCode dataset show that our approach outperforms the baseline CodeLlama model, achieving significant improvements in compilation success rates and functional correctness across Java, Python, and Kotlin. The CodePatchLLM framework is available at `https://github.com/dsshay/CodePatchLLM`.

§1. Introduction

Code generation or program synthesis aims to automatically generate source code that adheres to a specified programming requirement, which is typically described in a natural language [1,2]. Recently, with the development of large language models (LLM), techniques based on LLMs [3–6] have demonstrated impressive ability in code generation. However, challenges persist with the use of generated code in complex systems [7–9,12], indicating a remaining gap in fully meeting user expectations.

In this context, learning from automatic defect detection tools demonstrates exciting potential to enhance the comprehension of complicated

Key words and phrases: large language models, static analyzer, code quality.

technical specifications and the quality of generated codes [10]. Feedback from compilation and execution results is instrumental in directly ascertaining the functional correctness of programs [11, 12]. Researchers have introduced leveraging compiler feedback from unit tests to guide the exploration of the output space of LLMs [13,14] using reinforcement learning techniques. In other words, authors fine-tuned the model so that the output program was built successfully and passed tests.

Nevertheless, optimizing LLMs for code generation via compiler feedback presents several challenges. First, the increasing complexity of human requests to LLMs often results in the generation of longer code sequences, and this worsens the final program quality [15,16]. Second, feedback solely from independent unit tests and a compiler is not enough to ensure reliability of such a program. Static analysis tools conduct more thorough source code checks than compilers, which usually only detect syntax errors [17].

Automated code generation (or program synthesis) has attracted much attention over the past few years [18] because of its potential to improve the productivity of developers, as well as to speed up the software development [19]. Companies that hastily implement generative solutions or succumb to the "AI hype" may face a potential increase in cybersecurity risk. Recent work [20, 21] highlights the importance of using robust implementations of generative AI in a business environment to mitigate such risks. The demand for reliable and secure code has never been higher.

To tackle these challenges, several approaches have been proposed, including filtering and repairing the non-executable synthesized programs [22], using energy-based generation models with execution constraints [23], and reinforcement learning (RL) fine-tuning mechanisms [12,14,30]. However, existing approaches are often tailored to a specific programming language (PL) or task, e.g., the work [30] is exclusively designed for program synthesis task in Python.

We introduce a new approach, illustrated in Fig. 1, combining results of program analysis and LLM for code generation.

Initially, the output of any pre-trained LLM tailored for code generation is transferred to the program analysis module, SAFe (Software Analysis Feedback). The module includes but is not limited to the use of a compiler and a static analyzer. It is assumed that it is possible to include other analysis tools such as DAST [54] or IAST [24]. In particular, SAFe analyzes the entire file generated by the program to identify potential vulnerabilities in the architecture of the solution that may occur when using external

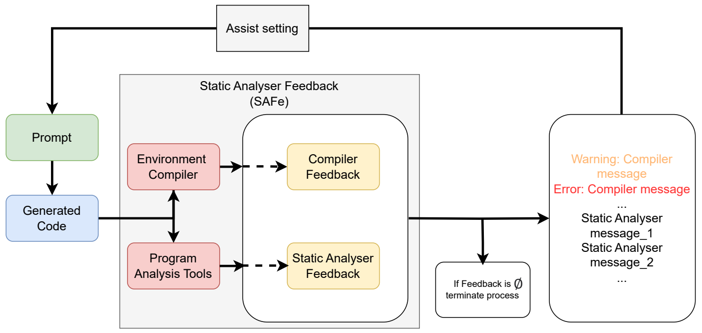


Figure 1. An overview of the proposed approach. Generated code is first initialized from the pre-trained LLM for the designed task and then transferred to the compiler and static analyzer (SAFe module) suitable for the selected programming language. Detected warnings and errors are collected in a single pool of messages that are transmitted to the assist setting. Finally, the LLM prompt is updated based on the obtained values and returned.

libraries and classes. The usual classic static analysis stages are performed on the generated program, namely capturing the program build to generate automatically the required intermediate representation, lightweight analysis of the program's syntax trees (AST-level analysis), and interprocedural dataflow analysis (which is both context-sensitive and path-sensitive based on symbolic execution). Eventually, the goal is to identify all possible errors in the generated program, before it is used by humans. One distinctive feature of this approach is the utilization of feedback from program analysis tools as additional contextual clues for the LLM.

Specifically, messages generated by the compiler and static analyzer are incorporated into the LLM's prompt as supplementary comments (in "Assist Setting"). By enriching the prompt with insights gathered from program analysis, the LLM gains a deeper understanding of the desired code's requirements, constraints, and potential pitfalls. Once the prompt is augmented with relevant analysis feedback, the LLM is prompted to generate a new code, embedding the provided comments into its output. This iterative process fosters a symbiotic relationship between automated program analysis and advanced language modeling, facilitating the generation of code that not only adheres to syntactic rules but also aligns with best practices, security guidelines, and architectural constraints.

In addition, we propose a novel framework, CodePatchLLM, which integrates the Svace static analyzer [48, 49] feedback into CodeLlama [6] to enhance the reliability and security of generated code. Through a series of experiments and evaluations on Leetcode dataset [55] we seek to demonstrate the effectiveness of our approach in improving code quality, reducing the risk of defects, and ultimately enhancing the trustworthiness of software systems in real-world applications.

To summarize, the major contributions of this paper are as follows:

- we introduce a novel approach that utilizes code-specific feedback as the external source of knowledge in model instructions. The approach is independent of models architecture and generates higher-quality codes;
- we develop the CodePatchLLM extension for CodeLlama that applies the Svace static analyzer to the generated code, and iteratively corrects the model's prompt using all warnings and errors detected by Svace;
- we demonstrate the effectiveness of CodePatchLLM through experiments across diverse programming tasks (from the Leetcode platform[1]) and program languages (Java, Python, Kotlin). Using CodeLlama with CodePatchLLM improves the compilation rate in 50% for Java and 10% for Kotlin more cases and functional correctness over different languages by 5%.

The remainder of this paper is organized as follows. In Section 2 we describe existing code generation models utilizing external knowledge and structure-based approaches. Section 3 delves into the specifics of our proposed approach and new CodePatchLLM framework that includes the Svace static analyzer. The experimental evaluation of CodePatchLLM on programming tasks written in three program languages (Java, Python, Kotlin) and the case study can be found in Section 4. Finally, Section 5 concludes the paper.

[1]`https://leetcode.com`

§2. Related work

2.1. Fine-tuning large language models for code generation. Recently, LLMs have shown remarkable ability in understanding natural language and code generation by training on large text corpora containing code data. Several pre-trained language models (PLMs) demonstrate significant potential for code generation including CodeGPT [18], PanGu-Coder [25], SantaCoder [26]. In addition, supervised fine-tuning models achieve more competitive performance such as CodeX [27], CodeLlama Instruct [6].

Reinforcement learning is a method of learning the optimal policy by exploring the environment and obtaining rewards [28]. Recently, some researchers have introduced RL to LLMs and improved the quality of the generated code by utilizing the unit test feedback to explore the output space of the policy model [12–14,29,30]. For instance, Coberly [30] leverages signal from unit tests as rewards and utilizes the actor-critic approach [31] to enhance models on code generation. PPOCoder [14] refines Code by employing the PPO algorithm [32] and RLTF [13] provides fine-grained rewards through the error locations, but the reward space is still sparse. However, the exploration of complex tasks in an environment characterized by a sparse reward is challenging. These methods still fall short of effectively using RL to enhance the model's performance in code generation [7].

2.2. Chain-of-thought prompting. With the recent advancements in large language models, researchers have discovered that utilizing the chain-of-thought (CoT) [33, 34] techniques can significantly improve reasoning abilities. The authors of [33] introduced the concept of few-shot CoT, which involves generating intermediate reasoning steps before arriving at the final answer with in-context demonstrations. This approach deviates from traditional few-shot prompting (also called in-context learning [35]) that directly generates the final answer. Zero-shot CoT [36] is another method leveraging chain-of-thought which adding the prompt "Let's think step by step" after the task description to activate LLMs to generate rationales in order for improved task performance. Other researchers also propose various prompting methods to enhance model capabilities, including auto-cot [37], least-to-more [38], decomposing prompting [39] and tree-of-thought [40]. In our work, outputs of program analysis tools can be seen as kind of prompt chaining, since all this data serves as intermediate steps for fixing bugs in the code.

2.3. Prompting with feedback. Despite the remarkable capabilities of large language models, it can still be challenging sometimes to generate the correct answer in a single attempt. Recently, people found that LLMs can receive feedback from external environment or generated by themselves and iteratively refine according to the feedback. Self-refine [41] launches a novel approach that allows LLMs to iteratively refine outputs with the feedback without any labeled data. Reflexion [42] proposes a "verbal reinforcement learning" that LLMs reflect on failures based on feedback and store reflexion in a text style for future trials. REMEMBERER [43] employs a method that allows LLMs to learn experience which is stored in an external memory from the feedback in the training set and transfer that experience to the test set for a better performance. In our work, we focus on code generation task and teach LLMs to improve code based on feedback from program analysis tools such as a static analyzer.

2.4. Prompting for code. Prompting techniques have been extensively utilized in tasks related to code. Some works including [44–46] focus on leveraging prompting to enhance code generation. In [47] prompting is used to facilitate code selection and develop a reviewer model. The main distinction between our work and them is that we focus on using program analysis tools to improve compilability and security of code without loss of efficiency in solving the problem, whereas all these previous works primarily rely on receiving execution results or error messages from the interpreter or using only a binary signal from the compiler.

§3. Method

In this section, we focus on the methodological details of our approach, which ensures the generation of a compiled program and a program tested by a static analyzer, respectively, as shown in Fig. 1. And we describe CodePatchLLM, a novel framework that uses Svace [48, 49] feedback for correcting prompt for CodeLlama [6], illustrated in Fig. 3.

3.1. Preliminaries. Let $s \in S$ denote a given input, which can be a piece of partial code, natural-language description, or a buggy program. Let $t \in T$ denote the generated source code. Formally, the problem of code generation can be formulated as learning a mapping f between the input space and target code space, i.e. $f : S \to T$. This paper explores the task of converting text into code across multiple programming languages, including Java, Python, Kotlin, utilizing identical input data.

Text-to-Code Generation. It aims to generate a whole program based on natural language description. Let $d = \{d_1, d_2, \ldots, d_{|d|}\}$ refer to a sequence of natural-language tokens. The text-to code generation task can be defined as generating source code $c = t \in T$, given the corresponding natural language description $d = s \in S$.

Feedback from a program analysis tool. As the whole program c is generated, even if it is a partial program, we feed it into a compiler to test whether it can be compiled successfully. Formally, we define the compiler feedback as

$$\text{feedback}_{\text{compiler}} = \text{Message}_{\text{compiler}}(c),$$

where the compiler feedback is a text (compiler message), and c denotes the code snippet fed into the compiler.

Similar to compiler feedback, we define the static analyzer feedback as

$$\text{feedback}_{\text{analyzer}} = \text{Message}_{\text{analyzer}}(c).$$

Messages from external code verification systems serve as an additional signal m for the program generation model c. These systems provide responses to the input code, which we denote as $m = \text{feedback}_{\text{compiler}} + \text{feedback}_{\text{analyzer}}$.

Moreover, there are two ways to utilize this data. First, it can be employed to design a reward signal that can be integrated with reinforcement learning [7,12,50]. Alternatively, it can be utilized as a clarifying comment for the model through prompt chaining.

Using external feedback as comment rather than solely relying on fine-tuning the model, we can explore more effective ways to improve code generation processes. In the next section, we consider some advantages and implications of prompting compared to fine-tuning.

3.2. Comparing prompting and fine-tuning approaches for code generation tasks. Below, we discuss several key aspects underlying the efficacy of prompting as a methodology over traditional fine-tuning practices. We begin by examining insights extracted from empirical studies [33, 39, 46].

Better understanding. Consistent improvement of parts of the generated code can facilitate better understanding, improved error correction, and more intensive optimization, ultimately leading to improved overall performance of the LLM [46].

Activating specific internal knowledge. Using prompting activates specific internal knowledge.

In the single step approach, the LLM is constrained to solve the entire problem in a single step. Therefore, the maximum number of tokens it can process is limited by s. In the n intermediary steps approach, the LLM solves the problem in multiple steps, with each step handling a portion of the task. Since the LLM can process tokens in each of the n intermediary steps, the total number of tokens processed across all steps is $s + n \times m$.

Comparing the two approaches, we can see that the n intermediary steps approach allows the LLM to process a total of $s + n \times m$ tokens, which is significantly higher than the token processing capacity of s in the single step approach.

Independence from the model architecture and software tools. Fine-tuning a model with a specific static analyzer can lead to overfitting, where the model performs well on errors and warnings that can only be detected by this static analyzer but poorly on unseen data.

Moreover, fine-tuning can sometimes lead to a loss of generalization ability, meaning the model becomes too specialized for the specific task it was fine-tuned for and performs poorly on related tasks or in different environments.

The transferability of a fine-tuned model to different tasks or domains may be limited. While fine-tuning may improve performance on a particular task, it might not transfer well to other tasks without further adjustments or retraining.

Customizing without computational cost. Fine-tuning can be computationally expensive, especially if the pre-trained model is large and the fine-tuning dataset is extensive. This can require significant computational resources and time, making fine-tuning less practical in some scenarios [51].

Previous works [11, 12] that fine-tune a model based on compiler feedback use an approach where a reward or binary signal is usually returned as feedback, and the context itself does not change. Thus, fine-tuning the model requires a lot of resources, since the model has to explore all the output spaces in order to find the best combination that will lead to a rare reward, program compilability.

External signals do not detract the model from solving the problem. If the previous code snippet is compilable, the generator can fool the compiler easily. Reinforcement learning is good at making use of this, resulting in the generated code can be compiled, but seriously deviating from the generation likelihood objective.

Previous works [12, 13] to avoid active model being too far away from reference model added a Kullback-Leibler [7] penalty with expectation. To alleviate the imbalance between the reward term and the penalty term and improve the stability of training, authors in [52] used autoregressive fine-tuning. This general setup leads to the fact that the original model corrects only a small number of tokens at the input, which in turn may not be optimal for quickly improving the compilability and security of the code, since the entire solution architecture has to be changed.

Based on this, we propose a simple and convenient framework CodePatchLLM that extends CodeLlama [6] capabilities for code generation through feedback from Svace [48].The flexible configuration of the framework allows to use any architecture of the LM with CodePatchLLM that solves the task. We believe that frameworks such as CodePatchLLM can be used as add-ons on an arbitrary code generation model and can be used when the specification requirement is higher than under normal conditions.

3.3. CodePatchLLM. Our proposed method CodePatchLLM enables large language models to use the code debugging method through a static analyzer. The CodePatchLLM can be used in cases where the requirements for the generated code are much higher than in normal conditions: the code must be vulnerability-tested and executable. It involves testing generated code through the Svace static analyzer. Programmers can analyze the output generated by the dialogue of LLM and Svace to understand how the code has been changed and what errors were in the first version.

Figure 3 shows an example of the first iteration using CodePatchLLM on the program generation task. We first let LLMs attempt solving the programming problem based solely on the problem description, without any extra information. The generated code, regardless of its size, is checked by the Svace static analyzer (Stage SAFe) and code regeneration based on feedback from program analysis tools (Stage Asssist Setting). The above steps will be repeated until Svace finds errors and vulnerabilities in the code or until several rounds of debugging attempts still fail to fix the issues.

Next we provide a detailed discussion of every step.

Generating code. For code generation we use Code Llama 70B Instruct[2], the largest and best-performing open-source model. We use a typical usage prompt for many models [34]:

> *You are a helpful and honest code assistant expert in* {LANGUAGE}*. Please, provide all answers to programming questions in* {LANGUAGE}.

where {LANGUAGE} denotes the selected programming language. When communicating with the model, we use the chat prompt format shown in Fig. 2.[3]

```
chat = [
    {"role": "system", "content": "System prompt    "},
    {"role": "user", "content": "First user query"},
    {"role": "assistant", "content": "Model response to first query"},
    {"role": "user", "content": "Second user query"},
]
```

Figure 2. Chat Prompt for CodeLlama-70B-Instruct.

SAFe stage In this stage, we proceed with code analysis that includes the running the Svace static analyzer. Following the Svace user manual, we initialize the directory in which the program file is located with the command: `svace init`. We capture any error messages provided by the Svace or the original compiler. It is important to note that even if the execution encounters errors, we still collect the log generated prior to the occurrence of the error. For instance, before running the analysis, Svace needs to build intermediate representation generated when monitoring the original compilation process (e.g. via the `javac` compiler for Java). And if an error occurs at the compilation stage, we collect the event and message in the log.

In our research, we use Svace as a base component for code analysis and optimization. Svace identifies security vulnerabilities inherent in software code, including potential buffer overflows, memory leaks, and other unsafe coding practices. By proactively detecting these vulnerabilities at an early

[2]CodeLlama-70B is free for research: `https://ai.meta.com/blog/code-llama-large-language-model-coding/`

[3]Source: `https://huggingface.co/codellama/CodeLlama-70b-Instruct-hf`

stage of development, Svace empowers users to mitigate security risks effectively, safeguarding software systems from potential cyber threats and breaches. Additionally, Svace facilitates the identification of performance bottlenecks within the codebase, such as inefficient algorithms, redundant computations, or memory-intensive operations.

It is assumed that the module SAFe can be flexibly expanded with other code analysis tools, for example, with a dynamic analyzer.

Assist settings. On the final step, we provide error messages to the model turning all feedback into a sequence. Our approach aims to take into account all feedback and instructions provided by program analysis tools. This means that at each timestep, the model can only utilize the past time steps data and itself. We instruct LLMs to regenerate the code considering the comments found, as illustrated in Figure 3. LLMs are prompted to fix an error in the code in a specific place (the line number and the method used are known). In Figure 3, two messages generated by the SAFe module are specified: a class declaration is expected for the compilation of the program and the calculation of the average value may be overflowed.

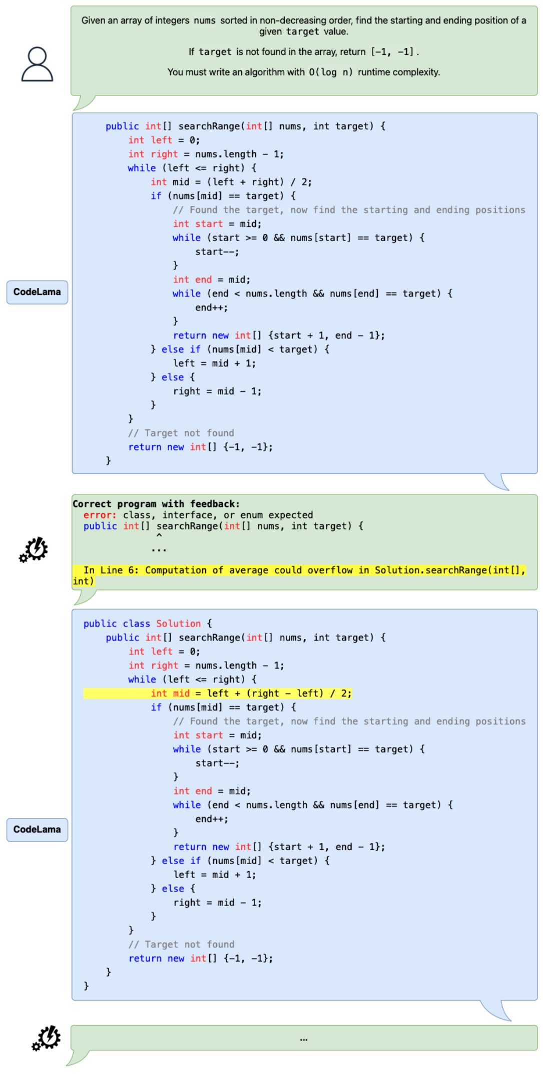
§4. Experiments

We evaluate our CodePatchLLM framework with CodeLlama [6] as the backbone. We investigate: (1) how CodePatchLLM framework can boost LLM performance on real-world programming tasks benchmarks, (2) the impact of the framework on code compilability and the reduction of defects, and (3) the effect of the number of iterations.

Benchmark. We consider Leetcode datasets [55] for our evaluations. LeetCode[4] is one of the most visited platform for practicing programming. We selected LeetCode as the primary source for our evaluation dataset, as it programming tasks can be directly compiled by copying and pasting contents from the LeetCode website. The dataset consists of 2 612 programming tasks. Problems in the dataset are categorized into three levels according to their difficulties.

Metrics. To evaluate the generated codes, we employ the *pass*@1 metric following [27], which calculates the percentage of problems for which all unit tests are passed using one synthetically generated program sample per problem.

[4]`https://leetcode.com`



Given an array of integers `nums` sorted in non-decreasing order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

You must write an algorithm with `O(log n)` runtime complexity.

```
    public int[] searchRange(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while (left <= right) {
            int mid = (left + right) / 2;
            if (nums[mid] == target) {
                // Found the target, now find the starting and ending positions
                int start = mid;
                while (start >= 0 && nums[start] == target) {
                    start--;
                }
                int end = mid;
                while (end < nums.length && nums[end] == target) {
                    end++;
                }
                return new int[] {start + 1, end - 1};
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        // Target not found
        return new int[] {-1, -1};
    }
```

```
Correct program with feedback:
  error: class, interface, or enum expected
  public int[] searchRange(int[] nums, int target) {
             ^
             ...

  In Line 6: Computation of average could overflow in Solution.searchRange(int[],
int)
```

```
public class Solution {
    public int[] searchRange(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                // Found the target, now find the starting and ending positions
                int start = mid;
                while (start >= 0 && nums[start] == target) {
                    start--;
                }
                int end = mid;
                while (end < nums.length && nums[end] == target) {
                    end++;
                }
                return new int[] {start + 1, end - 1};
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
        // Target not found
        return new int[] {-1, -1};
    }
}
```

...

Figure 3. Sample Java program generation with Code-PatchLLM, utilizing the Svace feedback.

Table 1. Performance results. Overall, CodePatchLLM boosts the performance and compilability of CodeLlama on the discussed metric.

		Accepted	**Wrong Answer**	**Runtime Error**	**Compiler Error**
Java	CodeLlama [6]	12%	22%	3%	62%
	Our	**25%**	**37%**	**5%**	**33%**
Python	CodeLlama [6]	36%	47%	17%	0%
	Our		same	result	
Kotlin	CodeLlama [6]	10%	2%	24%	64%
	Our	**12%**	**10%**	24%	**54%**

Implementation details. The experiment process was conducted on a device with three NVIDIA A100 80G GPUs.

The weights of the model are loaded from HuggingFace. The maximum output token length is set to 2048.

At each step of the code update, we submit the solution to the platform Leetcode. And also at each stage reinitialize existing Svace project directory from scratch.

4.1. Experimental results on Leetcode. In our study, we evaluate CodePatchLLM with Java, Python, and Kotlin languages.

The experimental results are illustrated in Table 1. The reported results indicate that the model with CodePatchLLM improve performance the model with basic settings for Java and Kotlin. And experiment shows that in Python the extended model does not improve the quality. Mechanism feedback from Svace and original compiler designed to executable and feasibility of the program. Therefore, in part, the extension does not change the overall quality, since a Python program does not require a compiler. The lack of feedback from the compiler and the analyzer explains the minimal deviations of the results from the original model. Experimental results show that CodePatchLLM improves executability by 45% for Java and by 10% for Kotlin. Additionally, it enhances the “Accepted” rate by 50% for Java.

In 37.3% of cases, CodeLlama incorrectly names the implemented class method at the first request. For example, `swapAdjacentNodes` instead

Table 2. Share of generated programs with errors and vulnerabilities. 98% of reviews have been corrected by CodePatchLLM.

	Compiler feedback	**Svace feedback**
Java	62,3%	12,5%
Python	—	0%
Kotlin	64,5%	3,1%

`swapPairs`. It is noteworthy that further dialogue with the model allows you to correct this although in the task formulation there are no clarifications in the naming of methods and classes.

Since CodePatchLLM constrains the model on an example when predicting another one, the model can simply "copy" the example without learning to understand the underlying task. In future, to address this, we can randomly mask between 0% and 5% of past tokens during training, which help regularize the model and prevent it from overfitting to the specific examples seen during training [29, 53].

Table 2 presents the percentage of generated programs with errors and vulnerabilities, categorized based on the type of feedback received from the original compiler and Svace static analyzer. the majority of the feedback (98%) has been addressed by CodePatchLLM, indicating its effectiveness in correcting errors. In Java, 12.5% of the generated programs had Svace feedback indicating vulnerabilities, while for Kotlin, this percentage was lower at 3.1%. No errors were detected in the generated Python programs, likely due to the Python support in Svace being in the first release, so that not many Svace checkers are supported for Python. It is worth noting that the analyzer verification stage comes after the compiler verification, and Svace identifies errors that do not influence whether the program can be successfully built, as all programs being analyzed are already compiled correctly. But the errors found can affect functional correctness.

Our approach allows to create a ready-made extension that can be used in conjunction with any LLM to generate code. Experiments with CodePatchLLM demonstrate that the method of using feedback from the original compiler and static analyzer improves compilability and prevents errors for Kotlin and Java programs.

§5. Conclusion

In this paper, we present CodePatchLLM, a novel framework that utilizes feedback from compilers and static analyzers to refine code generation prompts. Recognizing the limitations of fine-tuning LLMs for code generation, we developed a framework tailored specifically for programming languages rather than natural language. By integrating feedback from compilers and static analyzers, CodePatchLLM encourages the generation of syntactically and logically correct code. Our experimental results demonstrate that this approach significantly improves the syntactic and functional accuracy of generated code compared to the base model without CodePatchLLM. One limitation of CodePatchLLM is the increased computational cost due to the additional time required for data exchange. However, its primary advantage lies in enhancing pre-trained models' performance, which is more cost-effective than fine-tuning models for specific tasks.

This research addresses the pressing need for reliable code generation and has the potential to improve the development of more secure and dependable software systems.

References

1. A. Svyatkovskiy, S.K. Deng, S. Fu, and N. Sundaresan, *IntelliCode compose: Code generation using Transformer*, Proc. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (2020), 1433–1443.
2. S. Gulwani, O. Polozov, and R. Singh, *Program synthesis.* — Foundations and Trends in Programming Languages **4**, No. 1–2 (2017), 1–119.
3. D. Abulkhanov, N. Sorokin, S. Nikolenko, and V. Malykh, *Lapca: Language-agnostic pretraining with cross-lingual alignment*, Proc. 46th International ACM SIGIR Conference on Research and Development in Information Retrieval (2023), 2098–2102.
4. H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, M. Clement, R. Das, et al., *Llama 2: Open foundation and fine-tuned chat models*, ArXiv preprint arXiv:2307.09288 (2023).
5. A. Razzhigaev, M. Salnikov, V. Malykh, P. Braslavski, and A. Panchenko, *A system for answering simple questions in multiple languages*, Proc. 61st Annual Meeting of the Association for Computational Linguistics **3** (2023), 524–537.
6. B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X.E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, S. Sinha, et al., *Code llama: Open foundation models for code*, ArXiv preprint arXiv:2308.12950 (2023).

7. D.M. Ziegler, N. Stiennon, J. Wu, T.B. Brown, A. Radford, D. Amodei, P. Christiano, and G. Irving, *Fine-tuning language models from human preferences*, ArXiv preprint arXiv:1909.08593 (2019).
8. F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, L. Li, et al., *Pangu-coder: Program synthesis with function-level language modeling*, ArXiv preprint arXiv:2207.11280 (2022).
9. D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, et al., *Measuring coding challenge competence with apps*, ArXiv preprint arXiv:2105.09938 (2021).
10. Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, M. Lewis, et al., *Competition-level code generation with alphacode*. — Science **378**, No. 6624 (2022), 1092–1097.
11. X. Wang, Y. Wang, Y. Wan, F. Mi, Y. Li, P. Zhou, J. Liu, H. Wu, X. Jiang, and Q. Liu, *Compilable neural code generation with compiler feedback*, ArXiv preprint arXiv:2203.05132 (2022).
12. S. Dou, Y. Liu, H. Jia, L. Xiong, E. Zhou, J. Shan, C. Huang, W. Shen, X. Fan, Z. Xi, et al., *StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback*, ArXiv preprint arXiv:2402.01391 (2024).
13. J. Liu, Y. Zhu, K. Xiao, Q. Fu, X. Han, W. Yang, and D. Ye, *Rltf: Reinforcement learning from unit test feedback*, ArXiv preprint arXiv:2307.04349 (2023).
14. P. Shojaee, A. Jain, S. Tipirneni, and C.K. Reddy, *Execution-based code generation using deep reinforcement learning*, ArXiv preprint arXiv:2301.13816 (2023).
15. J. Hao, T. Yang, H. Tang, C. Bai, J. Liu, Z. Meng, P. Liu, and Z. Wang, *Exploration in deep reinforcement learning: From single-agent to multiagent domain*. — IEEE Transactions on Neural Networks and Learning Systems (2023).
16. P. Ladosz, L. Weng, M. Kim, and H. Oh, *Exploration in deep reinforcement learning: A survey*. — Information Fusion **85** (2022), 1–22.
17. B. Blanchet, P. Cousot, R. Cousot, J. Féret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, *A static analyzer for large safety-critical software*, Proc. ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (2003), 196–207.
18. S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, et al., *Codexglue: A machine learning benchmark dataset for code understanding and generation*, ArXiv preprint arXiv:2102.04664 (2021).
19. Md R. Parvez, W.U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, *Retrieval augmented code generation and summarization*, ArXiv preprint arXiv:2108.11601 (2021).
20. D. Humphreys, A. Koay, D. Desmond, and E. Mealy, *AI hype as a cyber security risk: The moral responsibility of implementing generative AI in business*. — AI and Ethics (2024), 1–14.

21. D.Y. Turdakov, A.I. Avetisyan, K.V. Arkhipenko, A.V. Antsiferova, D.S. Vatolin, S.S. Volkov, A.V. Gasnikov, D.A. Devyatkin, M.D. Drobyshevsky, A.P. Kovalenko, et al., *Trusted artificial intelligence: Challenges and promising solutions.* — Doklady Mathematics **106** (2022), S9–S13.
22. S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P.S. Liang, *Spoc: Search-based pseudocode to code.* — Advances in Neural Information Processing Systems **32** (2019).
23. T. Korbak, H. Elsahar, M. Dymetman, and G. Kruszewski, *Energy-based models for code generation under compilability constraints*, ArXiv preprint arXiv:2106.04985 (2021).
24. Y. Pan, *Interactive application security testing.* — 2019 International Conference on Smart Grid and Electrical Automation (ICSGEA) (2019), 558–561.
25. B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, et al., *Pangu-coder2: Boosting large language models for code with ranking feedback*, ArXiv preprint arXiv:2307.14936 (2023).
26. L. Ben Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C.M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, M. Dey, et al., *SantaCoder: Don't reach for the stars!*, arXiv e-prints (2023).
27. M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P. Pinto de Oliveira, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., *Evaluating large language models trained on code*, ArXiv preprint arXiv:2107.03374 (2021).
28. R.J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning.* — Machine Learning **8** (1992), 229–256.
29. H. Liu, X. Geng, L. Lee, I. Mordatch, S. Levine, S. Narang, and P. Abbeel, *Towards better few-shot and finetuning performance with forgetful causal language models*, ArXiv preprint arXiv:2210.13432 (2022).
30. H. Le, Y. Wang, A.D. Gotmare, S. Savarese, and S.C.H. Hoi, *CodeRL: Mastering code generation through pretrained models and deep reinforcement learning.* — Advances in Neural Information Processing Systems **35** (2022), 21314–21328.
31. V. Konda and J. Tsitsiklis, *Actor-critic algorithms.* — Advances in Neural Information Processing Systems **12** (1999).
32. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, ArXiv preprint arXiv:1707.06347 (2017).
33. J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q.V. Le, D. Zhou, et al., *Chain-of-thought prompting elicits reasoning in large language models.* — Advances in Neural Information Processing Systems **35** (2022), 24824–24837.
34. X. Hu, K. Kuang, J. Sun, H. Yang, and F. Wu, *Leveraging print debugging to improve code generation in large language models*, ArXiv preprint arXiv:2401.05319 (2024).
35. T. Brown, B. Mann, N. Ryder, M. Subbiah, J.D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., *Language models are few-shot learners.* — Advances in Neural Information Processing Systems **33** (2020), 1877–1901.

36. T. Kojima, S.S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, *Large language models are zero-shot reasoners.* — Advances in Neural Information Processing Systems **35** (2022), 22199–22213.
37. Z. Zhang, A. Zhang, M. Li, and A. Smola, *Automatic chain of thought prompting in large language models*, ArXiv preprint arXiv:2210.03493 (2022).
38. D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, et al., *Least-to-most prompting enables complex reasoning in large language models*, ArXiv preprint arXiv:2205.10625 (2022).
39. T. Khot, H. Trivedi, M. Finlayson, Y. Fu, K. Richardson, P. Clark, and A. Sabharwal, *Decomposed prompting: A modular approach for solving complex tasks*, ArXiv preprint arXiv:2210.02406 (2022).
40. S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, *Tree of thoughts: Deliberate problem solving with large language models.* — Advances in Neural Information Processing Systems **36** (2024).
41. A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al., *Self-refine: Iterative refinement with self-feedback.* — Advances in Neural Information Processing Systems **36** (2024).
42. N. Shinn, B. Labash, and A. Gopinath, *Reflexion: An autonomous agent with dynamic memory and self-reflection*, ArXiv preprint arXiv:2303.11366 (2023).
43. D. Zhang, L. Chen, S. Zhang, H. Xu, Z. Zhao, and K. Yu, *Large language models are semi-parametric reinforcement learning agents.* — Advances in Neural Information Processing Systems **36** (2024).
44. X.-Y. Li, J.-T. Xue, Z. Xie, and M. Li, *Think outside the code: Brainstorming boosts large language models in code generation*, ArXiv preprint arXiv:2305.10679 (2023).
45. J. Li, G. Li, Y. Li, and Z. Jin, *Enabling programming thinking in large language models toward code generation*, ArXiv preprint arXiv:2305.06599 (2023).
46. J. Li, S. Tworkowski, Y. Wu, and R. Mooney, *Explaining competitive-level programming solutions using LLMs*, ArXiv preprint arXiv:2307.05337 (2023).
47. T. Zhang, T. Yu, T. Hashimoto, M. Lewis, W.-t. Yih, D. Fried, and S. Wang, *Coder reviewer reranking for code generation.* — International Conference on Machine Learning (2023), 41832–41846.
48. V.P. Ivannikov, A.A. Belevantsev, A.E. Borodin, V.N. Ignatiev, D.M. Zhurikhin, and A.I. Avetisyan, *Static analyzer Svace for finding defects in a source program code.* — Programming and Computer Software **40** (2014), 265–275.
49. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, E. Velesevich, and D. Zhurikhin, *Design and development of Svace static analyzers.* — 2018 Ivannikov Memorial Workshop (IVMEM) (2018), 3–9.
50. R. Zheng, S. Dou, S. Gao, Y. Hua, W. Shen, B. Wang, Y. Liu, S. Jin, Y. Zhou, L. Xiong, et al., *Delve into PPO: Implementation matters for stable RLHF.* — NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following (2023).
51. M.A. Wiering and M. Van Otterlo, *Reinforcement learning.* — Adaptation, Learning, and Optimization **12**, No. 3 (2012), 729.
52. A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, and others, *Language models are unsupervised multitask learners.* — OpenAI Blog **1**, No. 8 (2019), 9.

53. N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A simple way to prevent neural networks from overfitting.* — The Journal of Machine Learning Research **15**, No. 15 (2014), 1929–1958.
54. M. Felderer, M. Büchler, M. Johns, A.D. Brucker, R. Breu, and A. Pretschner, *Security testing: A survey.* — Advances in Computers **101** (2016), 1–51.
55. W. Hou and Z. Ji, *A systematic evaluation of large language models for generating programming code*, ArXiv preprint arXiv:2403.00894 (2024).

Поступило 15 ноября 2024 г.

Ivannikov Institute for System Programming
of the Russian Academy of Sciences;
Moscow Institute of Physics and Technology
(National Research University);
HSE University,
Moscow, Russia
E-mail: `shaykhelislamov.ds@ispras.ru`

Ivannikov Institute for System Programming
of the Russian Academy of Sciences;
Moscow Institute of Physics and Technology
(National Research University);
ISP RAS Research Center for Trusted Artificial Intelligence,
Moscow, Russia
E-mail: `drobyshevsky@ispras.ru`

Ivannikov Institute for System Programming
of the Russian Academy of Sciences;
Moscow State University,
Moscow, Russia
E-mail: `abel@ispras.ru`