GPT4D: Automatic Cross-Version Linux Driver Upgrade Toolkit

Borui Yang[1], Hongyu Li[2], and Dongqi Cai[2]

State Key Laboratory of Networking and Switching Technology Computer Science Department
Beijing University of Posts and Telecommunications (BUPT)
{zbybr, lihongyu1999, cdq}@bupt.edu.cn

Abstract. The Linux operating system, with a history spanning over 22 years since its inception in 1991, has undergone countless version updates. Each update potentially introduces changes to its Application Programming Interfaces (APIs), compelling developers to adjust drivers in accordance with these API modifications. Consequently, developers often dedicate a significant amount of time to the maintenance and refactoring of existing code. As machine learning-based approaches have advanced, a growing number of code-specific models have been developed. However, most previous research on generative code models has predominantly concentrated on generating new code, frequently neglecting the unique requirements of editing existing code. In this paper, we propose a toolkit named GPT4D that harnesses the capabilities of the Large Language Model GPT-4 to automatically upgrade Linux driver code. A key challenge is the token limit of the model, by which most of driver code could be too long to be processed. To address this issue, we have designed a code filter that analyzes the abstract syntax tree (AST), based on code differences, to identify the most relevant code segments. We demonstrate the efficiency of GPT4D by applying it to a real-world Linux driver that requires modification.

Keywords: Software Engineering · Machine Learning · Programming Language.

1 Introduction

In the Linux kernel development community, we have observed that each Linux kernel version upgrade often involves modifications to its Application Programming Interfaces (APIs). These alterations can render Linux drivers relying on these APIs unusable. Throughout the development cycle of Linux kernel code [3], developers often invest a significant amount of time in code editing. Changes made to one part of the codebase typically have ramifications on many other sections, making manual propagation of these changes a labor-intensive and time-consuming endeavor. Consequently, we have embarked on exploring the feasibility of employing Large Language Models (LLMs) to automate code editing tasks.

EAI MLICOM 2023, December 17, Beijing, People's Republic of China
Copyright © 2024 EAI.SPRINGER-LNICST
DOI 10.4108/eai.17-12-2023.2342481

There are many toolkits developed based on Large-scale Language Models (LLMs), such as Github Copilot [1]. While these approaches effectively assist programmers in creating new code, they are not equally proficient at aiding in the revision of existing code and they do not track changes made by programmers and cannot anticipate where and how additional modifications should be made. Throughout the development cycle of a software project, developers often invest a substantial amount of time in editing code. Changes made to one part of the codebase typically have repercussions on many other parts, and manually propagating these changes can be a laborious and time-consuming task.

In recent years, there has been enormous interest in applying machine learning models for code generation, which has led to impressive performance on tasks such as program synthesis E. Nijkamp et al. [12], Y. Li et al. [9], program translation M. Lachaux et al. [8], M. Szafraniec et al. [15], type inference K. Jesse et al. [7], Wei et al. [16], and code auto-completion D. Guo et al. [6], A. Svyatkovskiy et al. [14], N. Nguyen and S. Nadi [10], F. Zhang et al. [18].

To solve this problem, initially we parse the code diff provided as input for LLM to identify keywords associated with the differences. In order to implement code modifications, it is crucial to incorporate the code files as an another part of the input. However, the entire code files are frequently surpass the maximum token limit of cutting-edge language models. Therefore, it is imperative to devise a strategy to tackle the challenge presented by these token limits. Subsequently, we elaborate on an algorithm, which including pre-order traversal of abstract syntax tree (AST) generating by the code file, for a static analysis method of code segments that may necessitate modification. Finally, we demonstrate the process of integrating all components to produce the ultimate editing outcome. The main workflow of GPT4D is in Fig. 1.

Contributions: In summary, we have made the following contributions.

- We propose a toolkit based on LLM to automate code editing work for Linux drivers.
- We design a code filter that analyzes the abstract syntax tree based on code differences to find the most relevant code segments.
- We conduct experiments to demonstrate the efficiency of GPT4D on real Linux drivers.

2 Background

In this section, we have presented the background for this paper, which is rooted in the rapid updates of the Linux Kernel code and the growing capabilities of Large Language Models (LLMs) in handling code-related tasks. We have also discussed the work of various researchers in this domain, providing context for our own research direction.

2.1 Development of Linux Kernel

The Linux kernel, celebrated for its adaptability and open-source philosophy, plays a pivotal role in modern computing. Its development, initiated by Linus

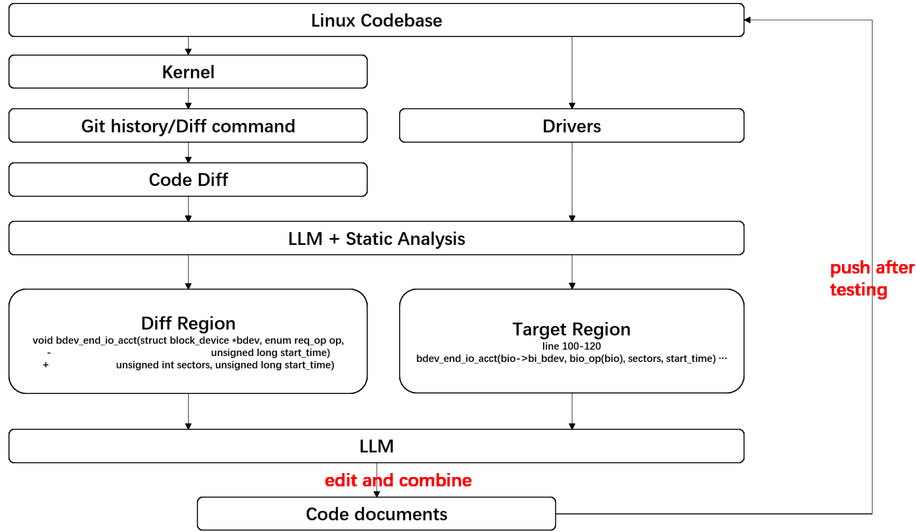

Fig. 1. The workflow of our driver upgrade toolkit. The user inspects the model output in each editing work and can optionally perform manual editing.

Torvalds in 1991 [3], has grown into a global collaborative effort, embodying the spirit of open-source innovation. A hallmark of the Linux kernel's evolution is its frequent updates, which are essential for introducing new features, enhancing security, and improving overall performance. However, these updates often bring changes to the kernel's Application Programming Interface (API). These changes, while crucial for progress, pose a significant challenge to existing device drivers. As the kernel undergoes iterative development, the APIs upon which drivers rely may evolve, leading to compatibility issues. Hence, with each update to the Linux kernel, developers are often required to manually modify driver code to ensure its compatibility with the new kernel version. This process entails meticulous adjustments to align the driver's functionality with the evolving kernel's structure and APIs. This emphasis the importance of automated code editing toolkits to reduce the possibility of making mistakes and the manual workload. This kind of toolkits become helpful for ensuring that drivers automatically remain functional and compatible with the latest Linux kernel versions, facilitating the seamless integration of new features and optimizations into the broader Linux ecosystem.

2.2 Using LLM for code editing works

In recent years, the field of software engineering has witnessed a shift in the way of code editing and maintaining, with the development of Large Language Models (LLMs). As software projects grow in complexity and scale, the manual editing of code becomes increasingly time-consuming and error-prone. LLMs,

exemplified by models like GPT-4 [2], have emerged as powerful toolkits capable of understanding and generating code that meets specific requirements set by their users. Recognizing the potential of LLMs in the domain of code editing, researchers and developers have explored their utility in automating a wide range of code-related tasks. This shift in perspective has been driven by the need for more efficient and accurately code editing, especially in the face of evolving codebases and the demand for rapid software updates. Using LLMs for code editing work is marked by the pursuit of enhanced developer productivity, code quality, and the automation of repetitive coding tasks, ultimately aiming to revolutionize the software development environment.

2.3 Related Work

In fact, many researchers are currently working on research related to automatic code editing. The past work similar to our setting is that of J. Wei et al. [17] and S. Brody et al. [4] which also focus on a contextual code editing. However, their researches primarily focuses on predicting the modifications to the remaining parts of the code based on given partial code changes within a single file. Both of their works, however, does not address the task of automatically code editing work under the premise that the provided context belongs to different files.

In S. Chakraborty et al. [5], they train a model to evaluate on future edits in the same codebase, but their work is non-contextual and depends on past code patches. Since the model does not condition on relevant changes, their technique requires retraining the model for new types of edit patterns.

Another work similar to ours is that of A. Ni et al. [11]. which focus on a code refactor based on API changes. But their entire design is targeted toward Python, which implies that it cannot be applied to Linux kernel code.

3 Methodologies

In this section, we delineate the primary workflow of our designed toolkit and the methods employed within it. Initially, we parse the code diff provided as input to identify pivotal keywords associated with the differences, including additions, deletions, replacements, and more. Subsequently, we expound upon an algorithm for the static analysis of code segments that may necessitate modification. Finally, we illustrate the process of integrating all components to produce the ultimate editing outcome.

3.1 Getting identifier from code changes

Firstly, we can obtain the code diff between different versions of the Linux kernel through system toolkits or GitHub History. Subsequently, we utilize this code diff as input, prompting LLM to identify the modified keywords, such as functions and variables. LLM then furnishes us with a list containing these identified keywords, which serves as input for the next subsection 3.2.

3.2 Analyzing Relevent Parts

Simply inputting the entire codebase as is would result in an excessive number of tokens, overwhelming the context. Instead, inspired by the ideas proposed in previous type inference work by M. Pradel et al. [13], we employ lightweight static analysis to extract the most relevant information as the context. Traditional direct retrieval methods are not dependable, as code cannot be treated like natural language; programming languages possess unique structural characteristics. Tree-sitter is a parser generator toolkit and an incremental parsing library renowned for its ability to construct a concrete syntax tree for a source file and efficiently update this tree as the source file undergoes edits. We employ Tree-sitter to parse the entire code file into an Abstract Syntax Tree (AST), subsequently traversing the AST to locate all 'identifier' and record their corresponding line numbers, creating what we term an 'identifier list.' We use the Algorithms1 below,

Algorithm 1 Get Identifier List

Input: AST;
Output: $identifierlist$;
1: **while** $Node$ is not the rightest node in AST **do**
2:   **if** type of $Node$ is identifier **then**
3:     $Node$ insert to $identifierlist$;
4:   **else**
5:     goto $Node->children$;
6:   **end if**
7: **end while**

After obtaining a 'differlist' and a 'identifierlist', the next step entails identifying which portions of the code are associated with these modified identifiers. We compared 'identifierlist' with the 'differlist' to pinpoint matching identifiers and associated line numbers, thereby yielding the relevant code segments. We use the Algorithms2 below,

Algorithm 2 Analyzing Relevent Parts

Input: $differlist$ and $identifierlist$;
Output: $relevantparts$;
for all $i = 0, 1, 2, \cdots, len(identifierlist) - 1$ **do**
2:   **if** name of $identifierlist_i$ in $differlist$ **then**
    $identifierlist_i$ insert to $relevantparts$;
4:   **else**
    continue;
6:   **end if**
end for

3.3 Code Editing by LLM

Providing a certain length of context is still necessary. This is because without context, the model cannot accurately perform the required edits. Therefore, we input the code diff block and the block to be processed, along with a specific prompt condition, to the LLM. The model then returns the modified result of the block to be processed. Of course, the LLM also assesses whether the block definitely needs to be modified, eliminating the need for us to design a separate method for this determination. Before this process, we need to record the scope of this code block in the original text to prevent incorrect final results due to repeated modifications. Finally, by replacing the modified code block in the original text, the overall code editing process is completed.

4 Motivating Example and Experiment

In this section, we first illustrate our technique using the example in Figure2 and we present experimental results that demonstrate why we should utilize GPT-4 and why it is necessary to reduce the length of the token sequence input to the model.

4.1 Motivating Example

In this subsection, we elucidate the functionality of GPT4D, as exemplified in Figure 2, which delineates the operational process of GPT4D. Subfigure (a) presents a code difference resulting from a Linux kernel upgrade, while subfigure (b) illustrates the identifiers that have been modified. Subfigure (c) exhibits the comprehensive identifier list derived from our static analysis method. Subfigure (d) subsequently displays the outcome post-LLM processing. We delve into a detailed analysis of this example in the following discussion.

Firstly, we procured a code difference resulting from a Linux upgrade on GitHub, which led to alterations in the API, as depicted in Subfigure (a). Initially, this code difference was utilized as input, which was subsequently fed into GPT-4. We presented a prompt to the model, instructing it to identify and return the modified identifiers list within this code difference as shown in subfigure (b), totally 4 iderntifiers was modified.

We utilized Tree-sitter to parse the modified code files, thereby generating an Abstract Syntax Tree (AST). A pre-order traversal of the entire AST was subsequently conducted. Given that Tree-sitter assigns a type to all identifiers, we saved the identifier and logged its line number if the attributes of the current node indicated it as an identifier during the traversal process. After this process, we can get a list illustrated in subfigure (c). Then we compared the 'difflist' and 'identifierlist' to identify the line numbers where all the keywords from the 'difflist' appeared. These lines are potentially in need of modification, so it's essential to record these line numbers in a list.

Additionally, we used some methods to fine-tune GPT-4, enabling it to produce more accurate and tailored results for code generation. As depicted in

Fig. 2. A demo of GPT4D.(a)A code diff as described below. (b)The list of identifiers that being modified. (c)Part of identifier list derived from preorder traversal of AST. (d)The result we get after the relevent part edited by GPT-4.

Figure 3, we set the temperature to 0 to prompt GPT-4 to provide more precise answers. Subsequently, we added our specifications for the desired output in the 'promote' field.

```
diff_prompt = "Please provide the changed functions based on the given code diff and output them in Python's list format."
diff_file = open('diff/diff.cpp', 'r')
diff_context = diff_file.read()
diff_code = diff_context
response = openai.ChatCompletion.create(
    model="gpt-4",
    temperature=0,
    messages=[
        {"role": "system", "content": diff_prompt},
        {"role": "user", "content": diff_code},
    ]
)
diff_list = ast.literal_eval(response.choices[0].message.content)
```

Fig. 3. An example of prompting GPT-4 to return content that better meets the requirements.

Finally, for every line 'x' necessitating modification, we supply GPT-4 with the code difference and several lines of code preceding and following line 'x' as context. We prompt the model to discern potential code editing tasks based on the code diff and code block, subsequently returning the processed code block to us. Upon receiving all the results, we substitute each code block with the corresponding section in the source file, thereby generating the final modified code file. Throughout the entire process, users can review the output results at any

point and even make modifications to achieve customizations. This significantly enhances the toolkit's flexibility.

4.2 Experiment

In this subsection, our primary emphasis is on addressing the following inquiries: the necessity of reducing the length of token sequence. We conducted an investigation into several advanced models trained using programming language code, as illustrated in Table1, to highlight their token limitations. In the examples provided in Section 4.1, the code file we needed to process had a total text length of 3404 lines. After the tokenization process, it contained approximately 50,000 tokens, significantly surpassing the maximum token limit. Our tests confirmed that these models were incapable of handling entire code files. Consequently, it was essential to perform code block parsing.

Upon utilizing our static analysis method to pinpoint the relevant code blocks, the quantity of tokens fed into the model is diminished to less than 1,000. Furthermore, we maintain ample context to guarantee that the Large Language Model (LLM) can accurately solve the task, subsequently generating suitable results.

Table 1. Max token limit of some language models.

Language model	GPT-3.5	GPT-4	GPT-4-32k	Llama2	CodeBERT-base	CodeT5
Token Limit	4096	8192	32768	4096	512	512

5 Conclusions

In this paper, we have developed a practical toolkit that leverages code differences generated by Linux kernel version upgrades. Using a Large Language Model(LLM), GPT4D automates the code modification of Linux driver, enabling it to adapt to new kernel versions, thereby eliminating the need for manual adjustments by community developers. This offers significant convenience for Linux driver maintainers. Moreover, we used a static analysis method to precisely pinpoint the parts of code that require modifications. This method drastically reduces the length of token sequence that needs to be processed by the Large Language Model. This solves a problem for the model from losing attention due to overly long token sequences, ensuring it performs as expected without affecting contextual understanding.

6 Limitations and Further Works

However, there are several limitations in our current work. As mentioned in section 4, firstly, one limitation of our work is that, at present, we still need to

use GPT-4 for our work. Secondly, the length of the code block we input into the LLM is fixed, but it is not optimal. Thirdly, as the Linux kernel version updates, there may be more complex or unpredictable code changes that hinder the normal operation of this toolkit. Lastly, our work has been conducted solely on the Linux operating system codebase. The universality of this toolkit for other C++ projects, or even projects in other programming languages (such as Python), still requires further development and experimental validation.

Accoring to what we mentioned above, a promising direction for our future work is to further develop this toolkit, we plan to expand it into IDE plugins, such as a VSCode extension. And we all know that programming languages are patterned languages. Although obtaining the modified keywords in the code diff through LLM has been proven to be a feasible method, we still hope to design a static analysis method to achieve this result. Furthermore, we are looking into extending the research methodologies utilized in this study to other codebases and even more programming languages. Our ultimate goal is to develop a more universally applicable and more accurate toolkit for automatic code editing toolkit that can potentially reduce manual efforts for developers in the future. Additionally, We did not design or tarin our own model, so we plan to train our custom model in the future to address similar issues.

References

1. Github copilot, https://github.com/features/copilot
2. Gpt-4 by openai, https://openai.com/research/gpt-4
3. Linux kernel codebase in github, https://github.com/torvalds/linux
4. Brody, S., Alon, U., Yahav, E.: A structural model for contextual code changes. Proceedings of the ACM on Programming Languages **4(OOPSLA)**, 1–28 (2020). https://doi.org/10.1145/3428283, https://doi.org/10.1145/3428283
5. Chakraborty, S., Ding, Y., Allamanis, M., Ray, B.: Codit: Code editing with tree-based neural models. IEEE Transactions on Software Engineering pp. 1–1 (2020). https://doi.org/10.1109/TSE.2020.3020502, https://doi.org/10.1109/TSE.2020.3020502
6. Guo, D., Svyatkovskiy, A., Yin, J., Duan, N., Brockschmidt, M., Allamanis, M.: Learning to complete code with sketches (2021)
7. Jesse, K., Devanbu, P., Sawant, A.A.: Learning to predict user-defined types. IEEE Transactions on Software Engineering (2022)
8. Lachaux, M.A., Rozière, B., Chanussot, L., Lample, G.: Unsupervised translation of programming languages. ArXiv (2020)
9. Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A.D.: Competition-level code generation with alphacode. arXiv preprint (2022)
10. Nguyen, N., Nadi, S.: An empirical evaluation of github copilot's code suggestions. In: Proceedings of the 19th International Conference on Mining Software Repositories. pp. 1–5 (2022)
11. Ni, A., Ramos, D., Yang, A.Z.H., Lynce, I., Manquinho, V., Martins, R., Goues, C.L.: Soar: A synthesis approach for data science api refactoring. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 112–124 (2021). https://doi.org/10.1109/ICSE43902.2021.00023, https://doi.org/10.1109/ICSE43902.2021.00023

12. Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: A conversational paradigm for program synthesis. arXiv e-prints pp. arXiv–2203 (2022)
13. Pradel, M., Gousios, G., Liu, J., Chandra, S.: Typewriter: Neural type prediction with search-based validation. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 209–220 (2020)
14. Svyatkovskiy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J.V., Allamanis, M.: Fast and memory-efficient neural code completion. In: 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). pp. 329–340 (2021)
15. Szafraniec, M., Rozière, B., Charton, H.L.F., Labatut, P., Synnaeve, G.: Code translation with compiler representations. ArXiv (2022)
16. Wei, J., Durrett, G., Dillig, I.: Typet5: Seq2seq type inference using static analysis. In: International Conference on Learning Representations (2023), https://openreview.net/forum?id=4TyNEhI2GdN
17. Wei, J., Durrett, G., Dillig, I.: Coeditor: Leveraging contextual changes for multi-round code auto-editing (May 2023), https://arxiv.org/abs/2305.18584
18. Zhang, F., Chen, B., Zhang, Y., Liu, J., Zan, D., Mao, Y., Lou, J.G., Chen, W.: Repocoder: Repository-level code completion through iterative retrieval and generation. arXiv preprint (2023)