PyFix

RYDER CATONIO* and JACK PARSONS*, University of Alberta, CAD

With the rapid development and popularity with large language models, there is an increasing availability of tools in the world that leverage these LLMs to help developers code more productively, but what if the same LLMs can be used to not just write code, but also automatically fix code containing bugs. That is the goal of this paper, to introduce a static analysis tool which leverages existing static analysis techniques, alongside LLMs, to find and fix bugs in python code.

The goal of this paper is to analyze the efficacy of using LLMs to solve bugs, given direct source code and a brief description of the problem. To get results for the goal, we introduce PyFix: a static analysis tool composed of an architecture with dual LLMs and a 5-stage pipeline to 1) use an existing static analysis tool to find bugs, 2) use the first LLM to propose a solution, 3) re-run step (1) to determine if the bug was fixed, 4) use the second LLM to validate and rank the effectiveness of the solution from step (2), 5) consolidate and pass off results to a developer, to analyse and act on them. Using PyFix we were able to successfully identify 15 static analysis problems with a success rate of approximately 40%.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: LLM's, Static Analysis, Python

ACM Reference Format:
Ryder Catonio and Jack Parsons. 2023. PyFix. *J. ACM* 0, 0 (2023), 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In the dynamic world of technology and software development, the progression of advancements continues to grow at an exponential rate. Demand is soaring for tools that empower developers to not only thrive in an ever-changing industry, but also ensure the production of high quality software. Furthermore, as seen in recent years, the emergence of large language models (LLMs) and the opportunities they present, has given rise to a whole new world of possibilities and fields of software.

While software practices for dynamic analysis such as unit testing and integration testing are vital to upholding software quality, the field of static analysis is also a pillar in producing software that is bug free, secure, and maintainable. This paper dives into a tool we have developed, that utilizes static analysis tools combined with LLMs to help alleviate much of the stress software engineers face. Our goal is to streamline and automate the process of finding and fixing bugs, allowing engineers to focus on much more critical tasks.

Our tool PyFix utilizes CodeQL, a querying language that can help find bugs in software ranging from simple issues such as unused variables or imports to cross-site scripting or SQL injection problems. The tool's flow involves first, loading the software you want to analyze. This is then ran

*Both authors contributed equally to this research.

Authors' address: Ryder Catonio, rcatonio@ualberta.ca; Jack Parsons, jparsons@ualberta.ca, rcatonio@ualberta.ca, University of Alberta, 116 St and 85 Ave, Edmonton, Alberta, CAD.

0004-5411/2023/0-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

against CodeQL queries to return results if the software is facing these problems. In this case, the data is then passed to a LLM such as GPT 3.5 or GPT 4.0 with a prompt to fix the issue. An LLM is a large language model, which receives textual input, with the goal of producing human-like output. LLMs are trained on massive amounts of data with the goal to generalize across many domains. After receiving the LLM's solution, the software is once again ran against the CodeQL query to verify that the problem is solved. Lastly, the LLM's proposed solution for the software is fed once again to a second LLM which is given a prompt to rank the solution based on the query results and how well it solved the issue. We will go into much more detail on the prompts used and ranking system later.

This tool hopes to shine light on the importance of utilizing static analysis tools in upholding software quality and also the strength LLMs have in alleviating some of the burden software engineers face.

2 OVERVIEW

In the pursuit of progressing the field of static analysis and embracing the capabilities of LLMs, our tool serves as a jump start to the endless possibilities of combining these two sub-fields of computer science. In this section, we will dive into the intricacies of PyFix, highlighting all the key concepts, tools, and the overall pipeline used.

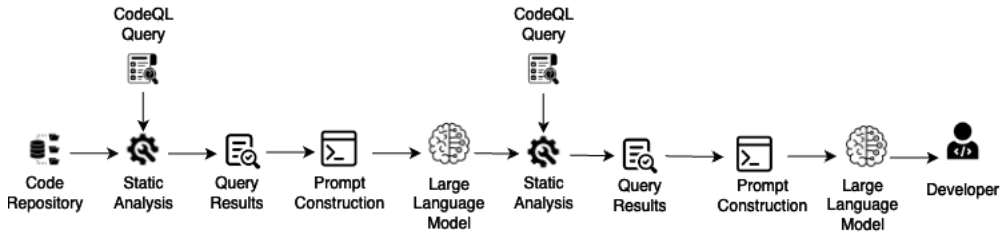


Fig. 1. PyFix Pipeline

2.1 General Flow

We will begin with an example shown below to help demonstrate the overall flow of the tool and also to help explain key concepts at each stage.

```
import random
from from glance.common import context

x = 0
if (x > 0):
    print("Yes")
else:
    print("No")
```

Fig. 2. Python code with an unused import

As we can see in figure 2, the program imports two modules: `random` and `context` that are never used within the program. Therefore, they are unneeded, lead to cluttered code, and can introduce performance impairments.

Beginning with the first stage of the pipeline we perform static analysis on the program by utilizing CodeQL and running queries against the code to detect if any problems arise. For example, running unused_import.ql [2] against this program returns “Import of ‘context’ is not used” and “Import of ‘random’ is not used”. As seen by the example, utilizing CodeQL can be an extremely powerful tool as the automation process is simple and the scalability of different queries is large.

After examining the results of the CodeQL query, the tool passes this data alongside the program to create a prompt for the solver LLM. We used the following prompt as a general template [9] when prompting the LLM to fix the problem

> "We are fixing code that has been flagged for the CodeQL warning titled *<query name>* which has the following description: *<query description>*
>
> The recommended way to fix code flagged by this warning is "query recommendation"
> Modify the Buggy code below to fix the CodeQL warning(s). Output the entire code block with appropriate changes. Do not remove any section of the code unrelated to the desired fix.
>
> The CodeQL warning(s) for the buggy code is: <CodeQL Results>
> The following input is the Python file containing the buggy code. Ensure that the output follows the same formatting and indentation conventions as the input code."

Fig. 3. Solver Prompt

As seen in the figure, we can see there are several key statements. Firstly, we pass the LLM the name of the query such as “Unused Import” and the description of the issue provided by CodeQL’s official website. Furthermore, we also pass the recommended general template for solving the issue. Lastly, we emphasize several times throughout the prompt to only modify lines of code that directly alter the program and to keep formatting and indentation conventions the same.

Following the first prompt construction we pass all this information to the LLM and receive back the python program with solver LLMs fix to the issue. Following alongside figure 2, we are given back

```
x = 0
if (x > 0):
    print("Yes")
else:
    print("No")
```

Fig. 4. Python code without an unused import

As you can see in figure 4, the LLM has proposed that we remove both imports as they are unnecessary.

Following receiving the code, we run this code against the CodeQL query again to check whether the query returns any results and if not, we once again create a new prompt [9], this time with

the ranking of the solution in mind. The following template is the general format we used for prompting the second LLM to rank the solution.

"You are an expert developer. You are verifying the code generated by LLM to fix the warning titled *<Query Name>* which has the following description: *<Query Description>*
The recommended ways to fix this code flagged for this warning are *<Query Recommendation>*

Score 0, if the patch has changes unrelated and unnecessary to fixing the warning (Strong Reject).
Score 1, if the patch has a few correct fixes, but still modifies the original snippet unnecessarily (Weak Reject).
Score 2, if the patch has mostly correct fixes but is still not ideal (Weak Accept).
Score 3, if the patch only makes edits that fix the warning with least impact on any unrelated segments of the original snippet (Strong Accept).

If you find additions or deletions of code snippets that are unrelated to the desired fixes (think LLM hallucinations), it can be categorically scored 0 (Strong Reject). That said, you can make exceptions in very specific cases where you are sure that the additions or deletions do not alter the functional correctness of the code, as outlined next.

Allowed Exceptions: The following (unrelated) code changes in the diff file can be considered okay and need not come in the way of labeling an otherwise correct code change as accept (score 2 or 3). This list is not exhaustive, but you should get the idea (a) deleting comments is okay, (b) rewriting a = a + 1 as a += 1 is okay, even though it may not have anything to do with the warning of interest, (c) making version specific changes is okay, say changing print ("hello") to print "hello".

The following (unrelated) code changes in the diff file are NOT considered okay, and you should label the diff file as reject (score 0 or 1) even if it is otherwise correct for the query. This list is not exhaustive, but you should get the idea (a) deleting or adding a print statement, (b) optimizing a computation, (c) changing variable names or introducing typos.
The following is the original file and the modified file with the fix to the problem. Output only the reason and score for the patch below. Do not output anything else."

Fig. 5. Ranker Prompt

Once again as seen in figure 5, we can see there are several key statements. We then pass the LLM the name of the query, the description of the problem, and the recommended general template for solving the issue [2].

The most important and core part of this prompt is the ranking used. As shown we ask the LLM to rank the solution based on the idea of whether or not the first LLM solved the problem in the most minimal way possible. The reason we do this is because often times LLM's struggle to listen directly to instructions so we want to rank the solutions based on its effectiveness to solve the issue without altering unrelated code. We allow certain exceptions that do not alter the logic behind the code such as a += 1 versus a = a + 1.

The final stage of the pipeline consolidates the original code, the *solvers* code, query results, the ranking, and the difference between the two programs. The data is then sent to the developer to review the results and make a decision whether the fix is up to code quality standard and does not introduce any new problems.

3 EXPERIMENTAL SETUP

3.1 Research Questions

Our central goal of this paper is to explore the intersection of static analysis with LLMs and see what they are capable of. We aim to see if LLMs have reached a point in development where they are capable of taking on extensive static analysis tasks and achieving similar performance to existing tools. Lastly, we wanted to see if the PyFix pipeline with dual LLMs is a capable system of solving various tasks.

3.2 Experimental Data

For evaluating PyFix, we analyzed results against 5 different static analysis problems that can be found using CodeQL queries:

1. Unreachable Code
2. Unused Local Variable
3. Unused Import
4. Module is Imported more than Once
5. Comparison of Identical Values

The CodeQL queries used are from the Python CodeQL suite [2]. Furthermore, for experimental python program data, we used the Codequeries hugging face dataset [8] to identify our static analysis programs to query the ETH Py150's [7] open dataset. We then condensed the results down to 15 program files, 3 for each of the static analysis problems.

3.3 Configurations

PyFix is currently compatible with any of OpenAI's LLMs [6] that are accesible through their API's. During our experiments, we ran PyFix using 1) GPT-3.5-turbo-1106/ GPT-3.5-turbo-1106, 2) GPT-4-1106-preview / GPT-4-1106-preview, 3) GPT-4-1106-preview/GPT-3.5-turbo-1106 for the solver and ranker LLM's respectively. Thus, through experiments we collected results for 45 different python programs using different configurations of OpenAI's LLMs

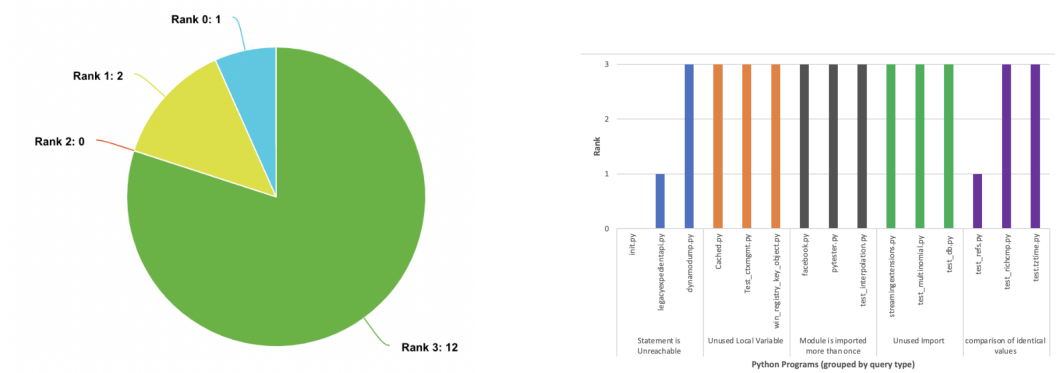
3.4 Evaluation

We evaluated PyFix using the ranks that the second LLM gave the solution to each individual python program. We then went and individually analyzed each of the solutions to determine how many of the ranks given were false positives, false negatives, and true positives. We based our analysis on whether us as developers would reject or accept the solution proposed. Note, some results may be ambiguous depending on which developer reviewed the solutions.

4 RESULTS

4.1 GPT 3.5 / GPT 3.5

When running experiments for the configuration of GPT 3.5 / GPT 3.5 for the solver and ranker LLMs respectively, can look at figure 6a and see 1 rank 0, 2 rank 1's, 0 rank 2's, and 12 rank 3's.



(a) GPT 3.5 vs GPT 3.5 Ranking Results

(b) GPT 3.5 vs GPT 3.5 Code Results

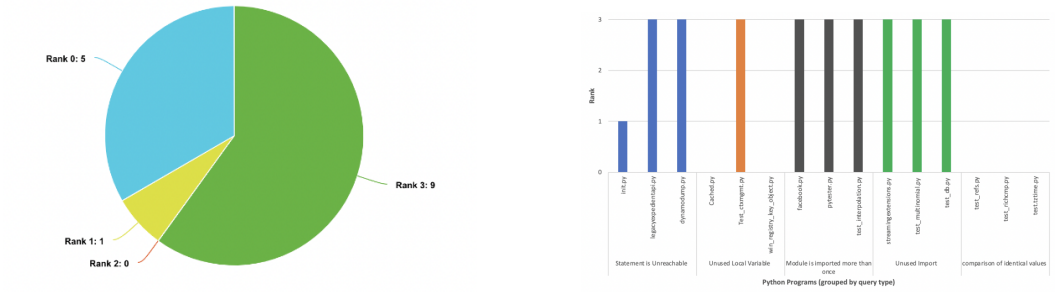
Fig. 6. GPT 3.5 / GPT 3.5 Data Results

Of the rank 3's, 7 of them resulted in false positives giving us 5 true positives. Therefore, using GPT 3.5 / GPT 3.5 gave us a success rate of approximately 33%.

Looking at figure 6b, all 5 queries resulted in rank 3's and each type also resulted in false positives. The true positives using this configuration came from modules being imported more than once, unused imports, and comparison of identical values.

4.2 GPT 4 / GPT 4

When running experiments for the configuration of GPT 4 / GPT 4 for the solver and ranker LLMs respectively, we got 5 rank 0's, 1 rank 1, 0 rank 2's, and 9 rank 3's.



(a) GPT 4 vs GPT 4 Ranking Results

(b) GPT 4 vs GPT 4 Code Results

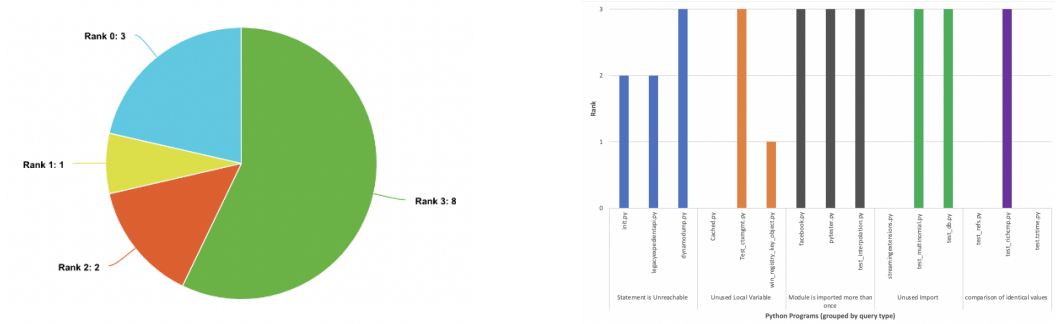
Fig. 7. GPT 4 / GPT 4 Data Results

Of the rank 3's, 2 of them resulted in false positives giving us 7 true positives. Therefore, using GPT 4 / GPT 4 gave us a success rate of approximately 46%.

Looking at the python programs grouped by query type, comparison of identical values was the only query type was that unsolvable by the first LLM and resulted in all 3 given a rank 0. However, one of those resulted in a false negative. Therefore, the only two false positives came from statements being unreachable.

4.3 GPT 4 / GPT 3.5

When running experiments for the configuration of GPT 4.0 / GPT 3.5 for the solver and ranker LLMs respectively, we got 3 rank 0's, 1 rank 1, 2 rank 2's, and 8 rank 3's.



(a) GPT 4 vs GPT 3.5 Ranking Results

(b) GPT 4 vs GPT 3.5 Code Results

Fig. 8. GPT 4 / GPT 3.5 Data Results

Of the rank 3's, 2 of them resulted in false positives giving us 6 true positives. Therefore, using GPT 4.0 / GPT 3.5 gave us a success rate of approximately 40%.

Comparision of identical values once again resulted in a false negative. The only query type that was left unsolved by the first LLM was when statements were unreachable.

5 LIMITATIONS

PyFix encounters a significant challenge when it comes to the solver LLM. As evident by our results, the solver LLM struggled severely when attempting to solve the static analysis problem within the program. Furthermore, both the solver and the ranker LLM showed difficulties in listening precisely to the prompt its fed. This led to extraneous results at certain points within our experiments, such as cutting out completely unrelated code.

Two additional limitations PyFix faces appears concerning financial costs and program size. Open AI's LLMs are serviced through charges each time their API is called. Thus the effectiveness of PyFix is directly tied to financial concerns and must be carefully considered when using the tool. Lastly, Open AI among other LLMs impose a max token limit, limiting the size of your prompts which impacts how large of a program you can pass it.

6 RELATED WORKS

In recent years as LLMs have continued to become more popular and more research has gone into them, there have been a vast increase in work related to PyFix.

Much of the work done to create PyFix must be accredited to Wadhwa et al. [2023] and their work in creating the pipeline used and the prompt engineering construction. Their work mirrors much of ours in an attempt to fix static analysis issues in Python using CodeQL and LLMs.

Many other researchers are conducting similar research into attempting to utilize LLMs to help automate static analysis tasks [3–5].

Catid's Hacks [2023] uses LLM's such as GPT, LLama, and Baize to detect C++ bugs and prompt the LLM to fix them. As well, Li et al. [2023] attempts to use open source LLMs to fix use-before-initialization bugs. They created the LLift framework which allows developers to interface with both static analysis tools and LLMs. Furthermore, Ding et al [2023] are attempting to leverage abstract syntax trees to perform static analysis to detect and fix python bugs.

Lastly, there are many other fields attempting to tackle the problem of solving static analysis problems. Other research we found [1] are conducting research into algorithms that fix static analysis problems based on pattern behaviour.

7 CONCLUSION

Static analysis is a complex and critical step in upholding software quality and work must be done to encorporate aspects in testing pipelines. However, static analysis can be time consuming and tedious and thus we created PyFix in hopes of eliminating the need of extra developer interaction. We ran PyFix against 45 different python programs and resulted in a true success rate of 40%.

Ultimately, a tool like PyFix will not work in current state of large language models. Even with the implementation of a multi-stage pipeline utilizing two large language models, the tool struggles to produce consistent results that are usable within a production codebase. The biggest flaw within PyFix remains that large language models ability to listen to the prompts its given. As well as it struggles to solve complex static analysis tasks. However, PyFix still remains a great tool when fed programs that are smaller in size and the CodeQL queries return limited results.

In the end, as companies such as OpenAI and many others continue to research, train, and improve these models, tools like PyFix can be forseen to be a critical asset in performing static analysis and increasing efficiency within the technology world.

REFERENCES

[1] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 159 (oct 2019), 27 pages. https://doi.org/10.1145/3360585

[2] CodeQL. [n. d.]. CodeQL Website. https://codeql.github.com/. Accessed: September 15, 2023.

[3] Hantian Ding, Varun Kumar, Yuchen Tian, Zijian Wang, Rob Kwiatkowski, Xiaopeng Li, Murali Krishna Ramanathan, Baishakhi Ray, Parminder Bhatia, and Sudipta Sengupta. 2023. A Static Evaluation of Code Completion by Large Language Models. (July 2023), 347–360. https://doi.org/10.18653/v1/2023.acl-industry.34

[4] Catid's Hacks. 2023. *Can open-source LLMs detect bugs in C++ code?* Retrieved October 23, 2023 from https://catid.io/posts/llm_bugs/

[5] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. (2023). https://doi.org/abs/2308.00245

[6] OpenAI. 2023. *OpenAI.* Retrieved December 8, 2023 from https://openai.com/

[7] ETH Py150. 2019. *ETH Py150 Open.* Retrieved December 8, 2023 from https://paperswithcode.com/dataset/eth-py150-open

[8] thepurpleowl. 2022. *Codequeries.* Retrieved December 8, 2023 from https://huggingface.co/datasets/thepurpleowl/codequeries

[9] Nalin Wadhwa, Jui Pradhan, Atarv Sonwane, Surya Parkash Sahu, Nagarajan Natarajan, Aditya Kanade, Suresh Parthasarathy, and Sriram Rajamani. 2023. *Frustrated with code quality issues?* Retrieved December 7, 2023 from https://paperswithcode.com/paper/frustrated-with-code-quality-issues-llms-can