**LLM Benchmark Projects**

**06 Sept, 2024**

1. Data C++: [PR2Commits\_perf\_issue\_C++ - Google Sheets](https://docs.google.com/spreadsheets/d/1PubFVYm6KShQ81B8ZqJq4lTmJ79djYhmYT5MehZFhF8/edit?gid=0#gid=0)
2. **Task-01:** Design a scrapper to separate codes
3. Pull request (PR) demonstrates a proposed code change
4. [JSON Lines](https://jsonlines.org/)
5. Scrap lib: [beautifulsoup4 · PyPI](https://pypi.org/project/beautifulsoup4/)

**07 Sept, 2024**

1. Generated 300 entries in jsonl file but need to check it with Prof if the format is desired

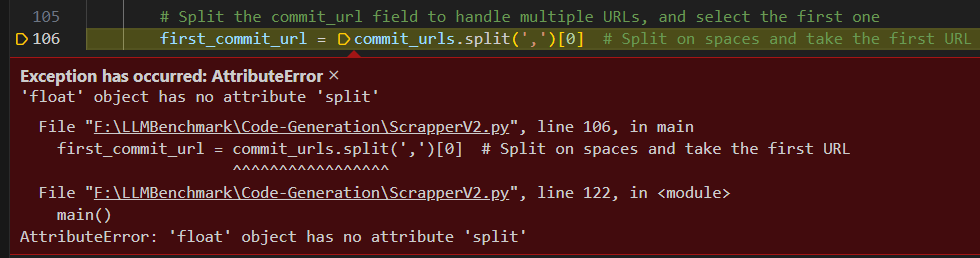
**09 Sept, 2024**

1. Meeting minutes: This is a new dataset: <https://docs.google.com/spreadsheets/d/1qDVTIzfJw70eB6KU61GV_JBFrMilPelAXaakvZz4350/edit?gid=0#gid=0>
2. Task-01: Need to verify this dataset if all the entries contain only 01 function change
3. Task-02: Design a scrapper on this dataset. Output entries: Commit title, Code difference (+,-), Before version, after version

**10 Sept, 2024**

1. If you're working locally, you can run git blame on the file to see which function the changes belong to:
2. git blame dbms/src/Functions/modulo.cpp -L 65,75
3. This command will show which function each modified line belongs to.
4. You can also use git log -p to inspect all changes over time and see how many functions have been modified over a series of commits.

**11 Sept, 2024**

1. Data parsing is success
2. Some error in a specific excel row
3. 
4. Solved

**14 Sept, 2024**

1. New paper: <https://dl.acm.org/doi/pdf/10.1145/3213846.3213874>
2. Paper’s code: <https://github.com/carolemieux/perffuzz>
3. In C++, **pathological input** refers to input data that causes a program to behave inefficiently or unexpectedly, typically due to the algorithm or data structure being used. This term is often used in the context of algorithm analysis and refers to inputs that trigger the worst-case performance of an algorithm.
4. The paper titled **"PerfFuzz: Automatically Generating Pathological Inputs"** introduces **PerfFuzz**, a tool designed to automatically generate inputs that trigger **pathological behaviors** in software, such as worst-case algorithmic complexity. The main objective is to discover inputs that cause the program to execute inefficiently, focusing on identifying performance bottlenecks.
5. **Fuzz testing** (or **fuzzing**) is a software testing technique that involves automatically feeding a program with large amounts of random, unexpected, or malformed inputs in an attempt to find vulnerabilities, crashes, or bugs. It is widely used for testing the robustness, security, and reliability of software.
6. Our goal is to generate inputs that independently maximize the execution count of each edge in the control-flow graph (CFG) of a program. We assume that we have one or more seed inputs to start with. These seeds are test inputs designed to verify functional correctness of the program and need not expose worst-case behavior. In our experiments, we use at most 4 seeds, but usually only 1.
7. A **Control Flow Graph (CFG)** is a graphical representation of all the possible paths that can be taken through a program during its execution. Each node in the graph represents a basic block, which is a straight-line code sequence with no branches (except at the entry and exit). The directed edges represent the control flow between these blocks, indicating the possible paths that the execution can take from one block to another.
8. if (x > 0) {

x = x + 1;

} else {

x = x - 1;

}

1. [Start]

|

[x > 0 ?]

/ \

[x = x+1] [x = x-1]

\ /

[Continue]

1. <https://www.geeksforgeeks.org/software-engineering-control-flow-graph-cfg/>
2. Performance testing? Generate Test cases. How can we generate test cases with AFL++?
3. Incorporate feedback with AFL++.
4. We collected some data from github. We’ll now generate test cases from AFL++ as manual test case generation will take too much time.
5. Then, we’ll test that chatGPT-generated code with these inputs either they are performing OK or not.
6. [AFLplusplus/AFLplusplus: The fuzzer afl++ is afl with community patches, qemu 5.1 upgrade, collision-free coverage, enhanced laf-intel & redqueen, AFLfast++ power schedules, MOpt mutators, unicorn\_mode, and a lot more! (github.com)](https://github.com/AFLplusplus/AFLplusplus)
7. [Building | AFLplusplus](https://aflplus.plus/building/)
8. [AFL++ : Combining Incremental Steps of Fuzzing Research | USENIX](https://www.usenix.org/conference/woot20/presentation/fioraldi)

**18 Sept, 2024**

1. WSL: [WSL 2: Getting started (youtube.com)](https://www.youtube.com/watch?v=_fntjriRe48&list=PLhfrWIlLOoKNMHhB39bh3XBpoLxV3f0V9)
2. Install: wsl –install
3. Then reboot
4. After the reboot, Ubuntu will be auto installed
5. After installing WSL, open your Linux terminal (like Ubuntu) and install make

sudo apt update

sudo apt install build-essential

1. **sudo apt install vim build-essential git clang**
2. **cd AFLplusplus**
3. **make**
4. **sudo make install**
5. UNIX username: reza\_fsm
6. Pass: reza123
7. If I search Ubuntu, then a new command shell will come
8. Wsl -l -v : to see the version
9. A Linux icon has appeared in file system
10. Make a c program
11. afl-gcc -g -fsanitize=address dvcp.c -o dvcp
12. ./dvcp
13. mkdir input
14. echo "test" > input/testcase.txt
15. afl-fuzz -i input -o output -m none -- ./dvcp @@
16. Vulnerable code: [hardik05/Damn\_Vulnerable\_C\_Program: An example C program which contains vulnerable code for common types of vulnerabilities. It can be used to show fuzzing concepts. (github.com)](https://github.com/hardik05/Damn_Vulnerable_C_Program)